Análisis de terminación para una plataforma de verificación de programas

Analysis termination for a program verification platform

Jakub Holubanský
Álvaro Mínguez

Grado de Ingeniería Informática
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de fin de grado

Madrid, 25 de agosto de 2016

Director: Ricardo Vicente Peña Mari

Índice general

Capítulo 1

Introducción

1.1. Inglés

In this project, we address the problem of how to automatically analyze the termination of programs. The context of the project is the assisted verification platform called CAVI-ART[1], which is being developed in the Informatic Systems and Computation Department under the grant TIN2013-44742-C4-3-R, funded by the *Ministerio de Economía y Competitividad*, and whose main researcher is the supervisor of this Graduation thesis.

The above platform will allow to verify programs written in different programming languages, both imperative and functional ones, and with the aim that most of its activities be independent of each particular source language, it translates programs into an Intermediate Representation (abbreviated IR) common to all of them.

The termination analysis, objective of this thesis, will receive as input a program written in the IR, and must determine whether the program terminates or not.

The first part of the thesis consists of a review of the techniques that have been proposed in the literature in order to analyze program termination, a problem which has been widely studied. We briefly present each technique and we discuss their advantages and limitations. Finally, we adopt one of them, which we have found toh be the most suitable to serve as a basis for our analysis. The chosen approach provides a tool, called RANK, which takes as input a particular kind of automaton and decides whether the automaton admits infinite traces ot not, that is whether it terminates or not. If so, it also provides a *rank function*, which is an expresion in the automaton variables which strictly decreases at each transition.

Our work has consisted of developing an algorithm which generates appropriate automata for RANK from programs written in the IR, in such a way that if RANK determines that the automaton terminates, so it does the IR program. The task has not been particularly easy having in mind that the IR is a functional language, i.e. programs have not an internal state, while RANK automata have an internal state represented by a set of integer variables, which are modified at every transition. On the other hand, the functional notation is recursive while the automata only support iterative loops.

Chapter 3 explains the details of using RANK and our way of encoding recursion into the automata, which at this point they are still generated by hand. Once convinced that the approach works, in Chapter 4 we explain the IR and our algorithm to generate automata from the IR. In both chapters, we give many examples of iterative and recursive programs.

1.2. Español

En este trabajo se aborda el problema de cómo analizar automáticamente la terminación de los programas. El contexto del mismo es la plataforma de verificación asistida llamada CAVI-ART[2], que está siendo desarrollada en el Departamento de Sistemas Informáticos y Computación, mediante el

[1] the full name is *Computer-Aided ValidatIon by Analysis, annotation, pRoof and Testing.*

[2] Las siglas provienen del nombre completo *Computer-Aided ValidatIon by Analysis, annotation, pRoof and Testing.*

proyecto TIN2013-44742-C4-3-R, financiado por el Ministerio de Economía y Competitividad y cuyo Investigador Principal es el director de este TFG.

La plataforma permitirá verificar programas escritos en diferentes lenguajes fuente, tanto imperativos como funcionales, y con el fin de que la mayor parte de sus actividades sean independientes de cuál es el lenguaje fuente utilizado en cada caso, traduce los programas a una representación intermedia (*Intermediate Representation*, abreviada IR) común para todos ellos.

El análisis de terminación objeto de este TFG recibirá como entrada un programa escrito en la IR, y deberá determinar si dicho programa termina o no.

La primera parte del trabajo realiza un revisión de las técnicas que se han propuesto en la literatura para analizar la terminación de los programas, problema este que ha sido muy estudiado. Se presenta brevemente cada una de ellas y se discuten sus ventajas y limitaciones. Finalmente, se opta por la que nos ha parecido más adecuada como base de nuestro análisis. La técnica elegida proporciona una herramienta llamada RANK, que toma como entrada un tipo particular de autómata y decide si dicho autómata contiene o no trazas infinitas, es decir si termina. En caso afirmativo, además suministra una *función de rango*, que es una expresión formada con las variables del autómata que decrece estrictamente en cada transición del mismo.

Nuestro trabajo entonces ha consistido en idear y desarrollar un algoritmo que genera un autómata apropiado para RANK a partir de un programa cualquiera escrito en la IR, de tal forma que si RANK determina que el autómata termina, eso implique que el programa de la IR también termina. La tarea no ha sido fácil teniendo en cuenta que la IR es un lenguaje funcional, es decir sin estado interno, mientras que los autómatas de RANK tiene un estado interno representado por un conjunto de variables enteras que se modifican en cada transición. Por otro lado, la notación funcional es recursiva mientras que los autómatas solo tienen bucles iterativos.

El Capítulo 3 explica los detalles de uso de RANK y nuestra forma de codificar la recursión en los autómatas, que de momento generamos manualmente. Una vez convencidos de que el enfoque funciona, en el Capítulo 4 explicamos la IR y nuestro algoritmo de generación de autómatas a partir de la IR. En ambos capítulos se dan numerosos ejemplos de programas iterativos y recursivos.

Capítulo 2

Estado del arte

En el campo de la terminación de algoritmos se han ido desarrollando distintos métodos para la demostración de terminación. En este capítulo, haremos un recorrido por los métodos más destacables, explicando su funcionamiento, limitaciones y mostrando ejemplos de su uso.

2.1. Size-Change Termination

Presentado [4] en 2001 por C. Soon Lee y N. D. Jones, el principio de $Size - ChangeTermination$ (SCT) para una función de primer orden expresa la idea de que un programa termina para cualquier entrada, siempre y cuando cada secuencia de llamadas infinitas cause un descenso infinito en alguno de sus parámetros.

Mediante la aplicación de este principio obtenemos un autómata de Büchi y un algoritmo directo que trabajara con $size - change\ graphs$ (SCG) sin la necesidad de transformarlos en autómatas.

2.1.1. Definiciones

En cuanto a la semántica del programa:

- Un programa es un conjunto de funciones. Cada función se representa de la siguiente forma: $f(x_1, ..., x_n) = e^f$, donde e^f es el *cuerpo* de la función. De igual manera el conjunto de parámetros se representa por: $Param(f) = \{f^1, ..., f^2\}$.
  - Escribiremos $c : f \to g$ para representar una llamada de f a g etiquetada con el número c.
  - Escribiremos $cs : f \to g$ para representar una cadena de llamadas de la forma: $c_0 : f_0 \to f_1,\ c_1 : f_1 \to .... \to f_n$ tal que $f_n = g$.
  - Representaremos un estado con un par Función x Valor de la forma (f, v). De esta forma una transición quedaría representada de la siguiente forma: $c(f, v) \to (g, u)$

2.1.2. Metodología

El objetivo de los siguientes apartados es mostrar cómo a partir del código de un programa o su secuencia de llamadas podemos usar SCT para decidir si el programa termina para todas las entradas posibles. Trabajaremos con el siguiente código de ejemplo:

$$\begin{aligned} a(m,n) = {} & if\ m = 0\ then\ n+1\ else \\ & if\ n = 0\ \ then\ {}_{\mathbf{1}}a(m-1,\ 1) \\ & \qquad else\ {}_{\mathbf{2}}a(m-1,\ ({}_{\mathbf{3}}a(m,\ n-1))) \end{aligned} \tag{2.1}$$

En el código vienen marcadas las tres llamadas. Construimos ahora un grafo para cada una de ellas, en la cual mostramos los parámetros origen a la izquierda y los parámetros destino a la derecha. En caso de que un parámetro de destino dependa del valor de uno de origen, y el valor no aumente, dibujaremos $<$ si disminuye siempre, o $\leq$ si puede disminuir o mantener el valor.

$$G1, G2 : a \to a \quad \boxed{\begin{matrix} m < m \\ \text{n} \quad \text{n} \end{matrix}} \quad G3 : a \to a \quad \boxed{\begin{matrix} m \leq m \\ n < n \end{matrix}}$$

Definición 1. Una secuencia de llamadas $cs = c_1c_2...$ está bien formada si, para cada i, $c_i : \ f_j \to f_k$ y $c_{i+1} : \ f_k \to f_l$. Es decir, si la función destino de una llamada es la función origen de la siguiente.

Definición 2. Un *multipath* es una secuencia finita o infinita de grafos SCT. En él, un hilo o thread es una secuencia infinita de arcos.

- Un hilo en un *multipath* M, es un camino conectado de arcos (un ejemplo esta marcado en la representación en negrita).
- Un hilo es descendente si contiene al menos un $<$. Un hilo es infinitamente descendente si contiene infinitas apariciones de $<$.

Definición 3. $FLOW^w$ es el conjunto de todas las secuencias infinitas de llamadas bien formadas, en las cuales se comienza desde la función inicial. Es decir

$$FLOW^w = \{cs = c_1c_2... \in C^w \mid cs \ esta \ bien \ formada \ y \ c_1 : \ f_{initial} \to f_1\}$$

Definición 4. $DESC^w$ es el conjunto de todos los elementos de $FLOW^w$ en los que algún thread tiene una cantidad infinita de descensos.

Definición 5. Si $DESC^w = FLOW^w$, toda secuencia infinita de llamadas termina, por lo que el programa siempre termina.

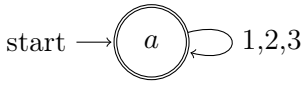
Definición 6. Un autómata de Büchi es un autómata no determinista que recibe cadenas infinitas. Se representa como

$$A = (In, S, S_0, \rho, F)$$

- In es el conjunto finito de caracteres de entrada.
- S es el conjunto finito de estados del autómata.
- S_0 es el estado inicial.
- ρ es el conjunto de transiciones de la forma $\rho \subseteq S \times In \times S$.
- F es el conjunto de estados finales.

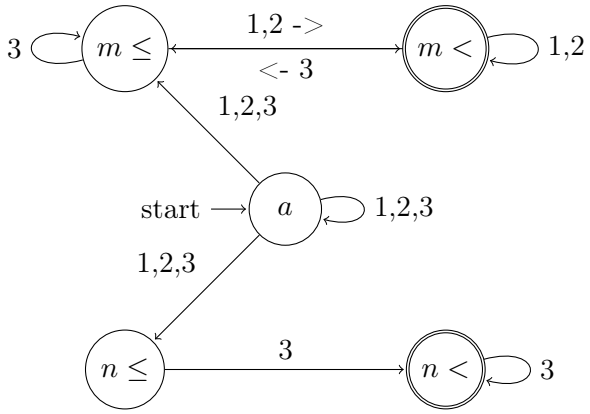
Este autómata acepta una cadena infinita si pasa un número infinito de veces por un mismo estado final. Lo usaremos en dos casos: para construir un autómata para $FLOW^w$, y un autómata para $DESC^w$. Una vez los hayamos construido, comprobaremos si reconocen el mismo conjunto de cadenas y, en caso afirmativo, podremos decir que el programa termina (pues en este caso $DESC^w = FLOW^w$).

- Para $FLOW^w$, construimos un autómata $A = (C, FcnName, \{f_{initial}\}, \rho, FcnName)$. El conjunto de transiciones ρ se forma como $\rho = \{(f, c, g) | c : f \to g\}$. Es decir, construimos un estado para cada función (la función inicial será el estado inicial), marcamos todos ellos como estados de aceptación y , para cada llamada en el programa, creamos una transición de la función origen a la destino con el número de llamada como carácter.



- Para el autómata de $DESC^w$, cogemos un *multipath* que describa la secuencia de llamadas del programa. Tras esto, para cada hebra infinita del *multipath*, realizamos el siguiente procedimiento:
  - Para cada arco de la hebra, han de existir estados representando los parámetros de destino, y el cambio que han sufrido ($<$ si el parámetro ha disminuido, o $\leq$ si ha sido copiado o disminuido).

- Representamos cada arco como una transición entre el estado del parámetro origen al estado del parámetro destino, cuyo parámetro además es el número de la llamada.
- Copiamos el autómata de $FLOW^w$ y, para cada llamada $c : f \to g$, creamos una transición (de parámetro c) desde f hasta los estados que representan los parámetros de g, con cambio $\leq$.



- Los SCG G1 y G2 (representados anteriormente) decrecen el valor de m en los arcos etiquetados con 1 y 2 respectivamente (al entrar en $m \leq$). SCG G3 decrece n y copia m (representado en los arcos etiquetados con 3).
- El ejemplo **a** termina puesto que no existe una secuencia infinita de transiciones que no decrementen m, eventualmente alcanzando el valor 0, decrementando n a cotinuación y retomando de nuevo el proceso.

Definición 7. El problema de comparar dos autómatas de Büchi se encuentra en el grupo de problemas PSPACE-completos, por lo que es un problema costoso. Esto se puede demostrar al haber un procedimiento para transformar un problema QBF (*Quantified Boolean Formula problem*, cuya complejidad está en el grupo de los PSPACE-completos) y reducirlo al de comparación de autómatas de Büchi.

2.1.3. Conclusión

Comparado con otros resultados, la terminación basada en principio de SCT es sorprendentemente simple y general: ordenes léxicos, llamadas indirectas a funciones y argumentos permutantes son automáticamente tratados sin ninguna modificación especial. En contraposición, se establece el problema de la complejidad intrínseca al método que resulta ser sorprendentemente alta, perteneciente al grupo de los problemas PSPACE-completos. Para nuestro trabajo en particular se pueden extraer conclusiones de este método, como la definición de hilo descendente e infinitamente descendente, pero como veremos más adelante existen métodos más efectivos.

2.2. Linear Arithmetic Simple While

El artículo nos muestra un método presentado por Andreas Podelski y Andrey Rybalchenko [6] para determinar la terminación de los programas clasificados como LASW (Linear Arithmetic Simple While), además de devolver una función de rango lineal para cada bucle en caso de que terminen. En los siguientes apartados explicaremos qué son estos programas y explicaremos el algoritmo usado para estos fines.

2.2.1. Definiciones

Definición 1. Un programa *LASW* es un programa que consta de un bucle **while** dentro del cual únicamente hay asignaciones, y cumple las siguientes características:

- El dominio de todas las variables es el de los enteros. Las llamaremos $x_1, x_2 ... x_n$

- Las condiciones del bucle están formadas por la conjunción de una o más inecuaciones lineales de la forma $c_1x_1 + ... + c_nx_n \leq c_0$.
- Las asignaciones de los valores se representan como inecuaciones lineales de la forma $a'_1x'_1 + ... + a'_nx'_n \leq a_1x_1 + ... + a_nx_n + a_0$, donde x_1 es el valor antiguo y x'_1 es el valor de la variable tras la asignación. Todas las asignaciones se ejecutan a la vez.

Los programas LASW siguen el siguiente esquema:

while ($Cond_1$ **and** ... **and** $Cond_m$) **do**
Múltiples asignaciones lineales
od

Definición 2. Un *estado del programa* es el conjunto de valores de las variables del programa en un momento dado de la ejecución.

2.2.2. Algoritmo

Teorema 1. Un programa LASW se puede representar con el siguiente sistema de inecuaciones:

$$(AA') \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x'} \end{pmatrix} \leq \boldsymbol{b}$$

donde $\boldsymbol{x}$ y $\boldsymbol{x'}$ son los vectores que representan el estado del programa antes y después de una iteración del bucle, A y A' son matrices y $\boldsymbol{b}$ es un vector de constantes. Por ejemplo, transformamos el siguiente programa LASW en el sistema de inecuaciones que incluimos debajo de él:

while $(i - j \geq 1)$ **do**
$(i, j) := (i - Nat, j + Pos)$
... se convierte en el siguiente sistema de inecuaciones:

$$\begin{aligned} -i + j &\leq -1 \\ -i + i' &\leq 0 \\ j - j' &\leq -1 \end{aligned}$$

Asignamos ahora los valores de los vectores a partir de las inecuaciones anteriores:

$$A = \begin{pmatrix} -1 & 1 \\ -1 & 0 \\ 0 & 1 \end{pmatrix}, A' = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & -1 \end{pmatrix}, \boldsymbol{x} = \begin{pmatrix} i \\ j \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$$

Teorema 2. Un programa LASW, representado de la forma $(AA') \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x'} \end{pmatrix} \leq \boldsymbol{b}$, termina si existen dos vectores de racionales $\boldsymbol{\lambda_1}$ y $\boldsymbol{\lambda_2}$ positivos tales que se cumpla el siguiente sistema de inecuaciones:

$$\begin{aligned} \boldsymbol{\lambda_1}, \boldsymbol{\lambda_2} &\geq \boldsymbol{0} \\ \boldsymbol{\lambda_1} A' &= \boldsymbol{0} \\ (\boldsymbol{\lambda_1} - \boldsymbol{\lambda_2}) A &= \boldsymbol{0} \\ \boldsymbol{\lambda_2}(A + A') &= \boldsymbol{0} \\ \boldsymbol{\lambda_2 b} &< 0 \end{aligned}$$

Teorema 3. Para un programa LASW, una vez obtenidas $\boldsymbol{\lambda_1}$ y $\boldsymbol{\lambda_2}$, podemos obtener una función de rango ρ a partir de la siguiente fórmula:

$$\rho(x) = def \begin{cases} \boldsymbol{rx} & \text{si existe } \boldsymbol{x'} \text{ tal que } (AA') \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x'} \end{pmatrix} \leq \boldsymbol{b} \\ \delta_0 - \delta & \text{en caso contrario} \end{cases}.$$

donde $\boldsymbol{r} = \boldsymbol{\lambda_2} A'$, $\delta_0 = -\boldsymbol{\lambda_1 b}$ y $\delta = -\boldsymbol{\lambda_2 b}$.

2.2.3. Conclusiones

En comparación con el **Size-change termination**, este método es más potente: mientras que SCT únicamente informa de si ha detectado o no la terminación del programa, el método de Podelski y Rybalchenko es capaz de devolver una función de rango. De hecho, este método es completo para aquellos programas que admiten una función de rango lineal, lo cual es una innovación respecto a métodos más antiguos. Además, la resolución de ecuaciones lineales se puede realizar en tiempo polinómico, por lo que resulta mucho más eficiente (la complejidad del método del SCT se encuentra en PSPACE completo). Sin embargo, su gran desventaja es su limitación a los programas LASW, por lo que no resulta de ayuda si el programa no cumple sus restricciones.

2.3. Isabelle

Isabelle es [**isabelle**] uno de los demostradores automáticos de teoremas más conocidos que utiliza internamente un método simple para generar funciones de rango y detectar terminación. Su simplicidad es lo que nos resulta interesante pese a que no es tan potente como los métodos anteriormente estudiados y su complejidad, debido a recursiones mutuas, está en la clase de problemas NP-completos. Pese a todo tiene un ratio de detección de terminación elevado (en torno al 85 %).

2.3.1. El método

El método de isabelle se divide en 4 pasos, enumerados y explicados brevemente a continuación y en mas detalle en el siguiente apartado.

- Construir un conjunto de medidas de tamaño basadas en los tipos de argumentos de la función.
- Para cada llamada recursiva y para cada medida, tratar de probar un descenso local (que la medida se hace más pequeña en la llamada). Guardar los resultados de la prueba en una matriz.
- Utilizando la matriz del paso anterior, tratar de buscar una tupla lexicográfica de las medidas para probar la terminación. Si esta combinación existe se puede encontrar en tiempo polinomial.
- Construir el orden de terminación global, usando el contenido del paso anterior para cada descenso local, para mostrar que todas las llamadas recursivas decrecen dicho orden.

2.3.2. Funcionamiento de Isabelle

Generar funciones de medida

Partiendo del tipo τ del argumento de la función, generamos un conjunto $\mathcal{M}(\tau)$ de funciones de medida siguiendo el siguiente esquema:

$$\mathcal{M}(\tau) = \{\lambda x. \mid x \mid_\tau\} \text{ si } \tau \text{ es un tipo de datos inductivo}$$

$$\mathcal{M}(\tau_1 \times \tau_2) = \{m \circ fst \mid m \in \mathcal{M}\ (\tau_1)\} \cup \{m \circ snd \mid m \in \mathcal{M}\ (\tau_2)\}$$

$$\mathcal{M}(\tau) = \{\} \text{ si } \tau \text{ es un tipo variable o una función}$$

Probar descensos locales

Cuando una medida se puede demonstrar que decrece en una llamada recursiva dada, tenemos un descenso local. Para cada llamada y cada parámetro, Isabelle primero prueba un descenso estricto:

$$\wedge v_1 ... v_m \Longrightarrow m_j\ r_i < m_j\ lhs_i$$

Si esto falla, prueba esta versión en su lugar:

$$\wedge v_1 ... v_m \Longrightarrow m_j\ r_i \leq m_j\ lhs_i$$

Donde r_i es el argumento de la llamada, lhs_i el argumento en la parte izquierda de la ecuación, m_j la medida y $v_1 ... v_m$ los diferentes vectores.

Los datos de la prueba se guardan en una matriz M con $M_{i,j} \in \{<, \leq, ?\}$, , en la que las filas de la matriz representan las diferentes llamadas recursivas, las columnas las diferentes medidas, $<$ y $\leq$ significan una prueba de un descenso estricto y no estricto respectivamente y ? un fallo de ambas pruebas. Un ejemplo de matriz sería:

$$\begin{pmatrix} < & < \\ \leq & < \end{pmatrix}.$$

Encontrar órdenes lexicográficos

Nuestro objetivo de encontrar un orden lexicográfico, teniendo en cuenta que las filas de nuestra matriz representan las diferentes llamadas recursivas y y las columnas las diferentes medidas, se puede reformular de la siguiente manera: Reordenar las columnas en la matriz de tal manera que cada fila empiece con una secuencia de $\leq$ (que puede ser vacía), seguido de un $<$. El siguiente algoritmo puede usarse para encontrar la solución:

- Encontrar una columna con al menos un $<$ y ningún ? y seleccionarla como nuestra nueva primera columna.
- Borrar todas las filas en las que la columna seleccionada contiene un $<$ en primer lugar, puesto que puede considerarse "solucionada". Después borrar también la columna en si misma y continuar la búsqueda en la matriz resultante.
- La búsqueda tiene éxito cuando la matriz esta vacía.

Ejemplo:

- Partiendo de la siguiente matriz:

$$\begin{pmatrix} < & \leq & > \\ ? & < & ? \\ \leq & \leq & < \end{pmatrix}.$$

- Cambiamos la segunda columna por la primera:

$$\begin{pmatrix} \leq & < & > \\ < & ? & ? \\ \leq & \leq & < \end{pmatrix}.$$

- Puesto que cumple los requisitos, podemos eliminar la primera columna y la segunda fila:

$$\begin{pmatrix} < & > \\ \leq & < \end{pmatrix}.$$

Prueba de reconstrucción

El paso 4 consiste en la formalización del paso anterior. Este paso, para nuestros fines, no es interesante puesto que la terminación queda probada en el paso 3.

2.3.3. Conclusiones

Como ya comentamos en la introducción, Isabelle nos proporciona un método simple para probar la terminación de programas. Como carencias, su coste está en la clase de problemas NP-completos y su ratio de aciertos no es comparable a otras herramientas existentes. Por todo esto, pese a que consideramos la herramienta Isabelle digna de estudio, nuestra preferencia será la herramienta presentada en el siguiente capítulo.

2.4. Herramienta Rank

Rank será nuestra herramineta de elección para realizar el trabajo. En este capítulo explicaremos nuestros motivos.

Algoritmos previos basados en funciones de rango afines, solo eran aplicables a bucles simples o no eran completos en el sentido en que no garantizaban una función de rango, si es que existía.

2.4.1. Objetivo y contribuciones innovadoras

Alias y co. proponen [1] un algoritmo eficiente capaz de trabajar con funciones de rango multidimensionales, que pese a ser voraz, los autores demuestran que es completo. Además de esto, nos muestran cómo manejar las funciones de rango que generan para obtener cotas superiores de la complejidad del programa, todo esto con un coste polinomial. Además, las funciones de rango permiten acotar bucles con un número de iteraciones superior a la lineal.

2.4.2. Definiciones

Integer Interpreted Automata

Definición 1. Un programa está representado como un *Integer Interpreted Automata* (K, n, k_{init}, T) definido por:

- un conjunto finito K de *puntos de control*.
- n variables enteras representadas por un vector x de tamaño n.
- un *punto de control* inicial $k_{init} \in K$.
- un conjunto finito T de 4-tuplas (k, g, a, k'), llamado *transiciones*, donde $k \in K$ y $k' \in K$ son los *puntos de control* origen y el destino respectivamente, la *guarda* $g : Z^n \mapsto B = \{\text{true, false}\}$, es una formula lógica expresada con inecuaciones afines $Gx + \boldsymbol{g} \geq 0$, y la *acción* $a : Z^n \mapsto Z^n$ que asigna, a cada valuación del vector x, un vector x' de tamaño n, expresado por una expresión afín $\boldsymbol{x'} = A\boldsymbol{x} + \boldsymbol{a}$. A y G son matrices y $\boldsymbol{g}$ y $\boldsymbol{a}$ son vectores.

Definición 2. El conjunto de estados es K x Z^n. Un camino desde $(k_0, \boldsymbol{x_0})$ a $(k, \boldsymbol{x})$ es una secuencia $(k_0, \boldsymbol{x_0}), (k_1, \boldsymbol{x_1}), ..., (k_i, \boldsymbol{x_i})$ tal que: $k_p = k, \boldsymbol{x_p} = \boldsymbol{x}$ y para cada $i, 0 \leq i \leq p$, existe en T una transición (k_i, g_i, a_i, k_{i+1}) tal que $g_i(\boldsymbol{x_i}) = \text{true}$ y $\boldsymbol{x_{i+1}} = a_i(\boldsymbol{x_i})$.

Definición 3. Un estado $(k, \boldsymbol{x})$ es *alcanzable* si existe un camino desde el estado inicial $(k_0, \boldsymbol{x_0})$ a $(k, \boldsymbol{x})$. Al conjunto de los estados alcanzables lo llamamos $\mathcal{R}$.

Definición 4. Denotaremos $\mathcal{R}_k$ al conjunto de posibles valores de $\boldsymbol{x}$ cuando se alcanza el punto de control k.

$$\mathcal{R}_k = \{\boldsymbol{x} \in Z^n | (k, \boldsymbol{x}) \in \mathcal{R}\}.$$

Termination and Ranking Functions

Definición 5. Una función de rango, es una función $\rho : K \text{ x } Z^n \to W$ de los estados del autómata a con un conjunto bien fundado $(W, \leq)$, cuyos valores decrecen en cada transición $t = (k, g, a, k')$

$$\boldsymbol{x} \in \mathcal{R}_k \wedge g(\boldsymbol{x}) = true \wedge \boldsymbol{x'} = a(\boldsymbol{x}) \Rightarrow \rho(k', \boldsymbol{x'}) < \rho(k, \boldsymbol{x})$$

La existencia de una función de rango implica la terminación del programa para cualquier conjunto de valores $\boldsymbol{v}$ en el punto de control inicial k_{init}

A Greedy Polynomial-Time Procedure

Definición 6. La **envolvente convexa** de un conjunto es el menor poliedro que contiene a todos los puntos del conjunto.

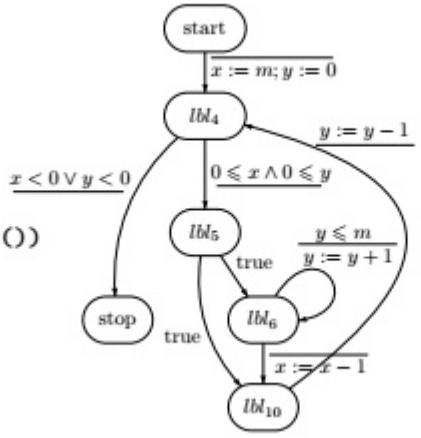
Definición 7. Llamamos $\mathcal{P}_k$ a la envolvente convexa del conjunto $\mathcal{R}_k$ ($\mathcal{R}_k \subseteq \mathcal{P}_k$), la cual representa al conjunto de invariantes de estado k.

Definición 8. Llamamos $\mathcal{Q}_t$ al poliedro que describe las restricciones de la transición t: $\boldsymbol{x} \in \mathcal{P}_{\|}$, $g(\boldsymbol{x})$ se cumple y $\boldsymbol{x'} = a(\boldsymbol{x})$.

2.4.3. Algoritmo

Usaremos el siguiente código en C como programa de ejemplo para nuestro algoritmo. En primer lugar, se transforma el código del programa a un *Integer Interpreted Automata*, el cual podemos ver en el centro. En caso de no ser posible transformar una guarda, la reemplazaremos por *true*. Tras esto, obtenemos las invariantes para cada estado del autómata usando, por ejemplo, la herramienta *ASPIC* [3] (usada por *RANK*).

```
y = 0;
x = m;
while(x>=0 && y>=0){
  if(indet()){
    while(y <= m && indet())
      y++;
    x--;
  }
  y--;
}
```



lbl_4	$m \geq x > 0, m \geq y > 0$
lbl_5	$m \geq x \geq 0, m \geq y \geq 0$
lbl_6	$m \geq x \geq 0, m+1 \geq y \geq 0$
lbl_{10}	$m \geq x \geq -1, m+1 \geq y \geq 0$, $2m \geq x+y$
start	$2m+4$
lbl_4	$(2x+3, 3y+3)$
lbl_5	$(2x+3, 3y+2)$
lbl_6	$(2x+2, m-y+1)$
lbl_{10}	$(2x+3, 3y+1)$
WCCC	$5+7m+4m^2$

Nuestro objetivo es conseguir la función de rango multidimensional $\rho = (\sigma_0, \sigma_1, ..., \sigma_d)$ del autómata, donde d es su dimensión. Para ello, construiremos un sistema de inecuaciones y usaremos un algoritmo de múltiples iteraciones en el cual obtendremos una σ_i en cada iteración. Comenzamos analizando los dos requisitos de las funciones de rango:

- Una función de rango multidimensional asigna un vector no negativo a cada estado relevante:

$$\boldsymbol{x} \in \mathcal{P}_k \implies \rho(k, \boldsymbol{x}) \geq \boldsymbol{0}$$

- El resultado decrece en cada transición. Para comparar los vectores resultado usamos el orden lexicográfico, por lo que un vector $\boldsymbol{a}$ se considera menor que otro vector $\boldsymbol{b}$ si, al recorrer de izquierda a derecha, el primer elemento en el que difieren es menor en $\boldsymbol{a}$.

$$(\boldsymbol{x}, \boldsymbol{x'}) \in \mathcal{Q}_t \implies \Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x'}) >_d \mathbf{0}$$

siendo $t = (k, g, a, k')$ cualquier transición y $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x'}) = \rho(k, \boldsymbol{x'}) - \rho(k, \boldsymbol{x})$.

En cada iteración construiremos una única σ_i de la función de rango, cambiamos la ecuación anterior para comparar la posición del vector correspondiente a la iteración actual. Además, expresamos la diferencia como ϵ_t:

$$(\boldsymbol{x}, \boldsymbol{x'}) \in \mathcal{Q}_t \implies \Delta_t(\sigma, \boldsymbol{x}, \boldsymbol{x'}) \geq \epsilon_t, \text{ donde } 0 \leq \epsilon_t \leq 1.$$

Aplicando el lema de Farkas [7] sobre la primera fórmula, podemos asegurar que:

$$\forall k \exists \boldsymbol{\lambda_k} \in (\mathbb{R}^+)^n, \boldsymbol{\lambda_k^0} \in \mathbb{R}^+ \text{ tales que } \sigma(k, \boldsymbol{x}) \equiv \boldsymbol{\lambda_k}(P_k \boldsymbol{x} + \boldsymbol{p_k}) + \boldsymbol{\lambda_{k0}}$$

... mientras que aplicándolo sobre la última obtenemos:

$$\forall t \exists \boldsymbol{\mu_t} \in (\mathbb{R}^+)^n, \boldsymbol{\mu_t^0} \in \mathbb{R}^+ \text{ tales que } \triangle_t(\sigma, \boldsymbol{x}, \boldsymbol{x'}) - \epsilon_t \equiv \boldsymbol{\mu_t} * (Q_t y + \boldsymbol{q_t}) + \boldsymbol{u_t^0}$$

Por tanto, obtenemos la siguiente inecuación para cada estado k:

$$\exists \boldsymbol{\lambda_k} \in (\mathbb{R}^+)^n, \boldsymbol{\lambda_k^0} \in \mathbb{R}^+ \text{ tales que } \boldsymbol{\lambda_k}(P_k \boldsymbol{x} + \boldsymbol{p_k}) + \boldsymbol{\lambda_{k0}} \geq 0$$

... y la siguiente inecuación para cada transición t:

$$\exists \boldsymbol{\mu_t} \in (\mathbb{R}^+)^n, \boldsymbol{\mu_t^0} \in \mathbb{R}^+ \text{ tales que } \boldsymbol{\mu_t}(Q_t \boldsymbol{y} + \boldsymbol{q_t}) + \boldsymbol{u_t^0} \geq 0$$

Antes de resolver el sistema de inecuaciones anterior necesitamos calcular los valores de P_k y $\boldsymbol{p_k}$ para cada estado (obtenibles a partir de las $\mathcal{P}_k$), y de Q_t y $\boldsymbol{q_t}$ para cada transición (obtenibles a partir de las $\mathcal{Q}_t$). Es decir, tenemos que obtener una matriz A y un vector $\boldsymbol{a}$ para cada uno de los poliedros anteriores. Para ello, seguimos el siguiente procedimiento:

- Para cada poliedro, construimos una matriz A de tamaño $m \times n$, donde m es el número de invariantes y n el número de variables.
- Representamos cada inecuación i, $0 \leq i < m$ del poliedro anterior en la forma $b_0 x + b_1 y + ... \geq c$.
- Añadimos una nueva fila $(b_0, b_1 ... b_{n-1})$ a la matriz A y el valor c al vector $\boldsymbol{a}$.

Por ejemplo, a continuación se muestran las matrices y vectores para *lbl6* y la transición *lbl6* $\rightarrow$ *lbl6*:

$$P_k = \begin{pmatrix} -1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix},$$

$$\boldsymbol{p_k} = (-m, -m-1, 0),$$

$$Q_t = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix},$$

$$\boldsymbol{q_t} = (-m, -m-1, 0, -m, -1, -1).$$

En el ejemplo anterior, observamos que la matriz Q_t consta de 6 filas: 3 para restringir los valores posibles a aquellos del estado k, 1 para representar la guarda de t y 2 para representar el nuevo valor ($y' = y + 1$ se representa como $-y + y' \geq 1$ y $y - y' \geq -1$). Una vez hemos obtenido los valores para las ecuaciones anteriores, pasamos a obtener la función de rango. Para ello, comenzamos con una ρ vacía y realizamos el siguiente proceso de forma repetida:

- Buscamos una función σ tal que el mayor número de ϵ_t tengan valor 1. Si no encontramos una, el algoritmo no encuentra una función afín multidimensional.
- Añadimos esa función como una nueva dimensión de la función de rango ρ, para cada invariante.
- Eliminamos todas las transiciones en las que ϵ_t fueran 1.
- Repetimos hasta haber eliminado todas las transiciones de la lista de ecuaciones.

Si hemos eliminado todas las transiciones, entonces ρ es una función de rango para el autómata y el programa termina. Este algoritmo tiene un coste polinomial y, si un autómata tiene una función afín multidimensional, ésta es encontrada (es decir, es completo).

2.4.4. Worst-Case Computational Complexity

Una vez obtenida la función de rango f, la herramienta *RANK* permite obtener, mediante manipulación algorítmica, un *Worst-Case Computational Complexity* o **WCCC** para un programa, cuyo valor representa una cota superior del número de transiciones del programa.

2.4.5. Conclusiones

El método descrito por Alias y compañía fue muy innovador en el mundo de la terminación de algoritmos, puesto que es capaz de obtener funciones de rango en una gran proporción de los programas en lenguaje C. Anteriormente habíamos visto el algoritmo *Size-Change Termination*, el cual también era aplicable a una gran proporción de programas. Sin embargo, su complejidad se encontraba en el grupo de los problemas $PSPACE-completos$ y no obtenía funciones de rango, por lo que el coste del nuevo algoritmo (polinómico) es un gran avance para este aspecto ($LASW$ restringía mucho los programas analizables). *Transition Invariants* era también prometedor en este aspecto, sin embargo no se llegó a conseguir su aplicación en la práctica. Además, la inclusión del $WCCC$ permite no sólo conocer la terminación de un programa, sino también tener una idea del tiempo de ejecución aproximado, lo cual no hace ninguno de los anteriores. Por todas estas razones hemos decidido utilizar este enfoque y la herramienta RANK para probar la terminación de programas en la plataforma CAVI_ART,

Capítulo 3

Metodología y uso de RANK

Al final del capítulo anterior explicamos RANK, un método para detectar la terminación de algoritmos que obtiene funciones de rango multidimensionales en tiempo polinomial. Esto lo hace destacar sobre el resto de métodos que hemos visto, y por ello será el que usaremos a la hora de analizar ejemplos. Comenzaremos este capítulo con instrucciones para su uso y acto seguido mostraremos ejemplos de su aplicación a algoritmos conocidos.

3.1. Instrucciones de uso

3.1.1. Formato de entrada

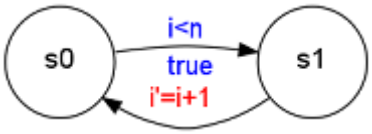
Como mencionamos en el capítulo anterior, RANK soporta como entrada un *Integer Counter Automata* en formato [2]. Este formato consta de dos secciones:

- Una sección **model**, en donde se especifican las variables, estados y transiciones del autómata. Para los dos primeros indicaremos simplemente sus nombres, mientras que en las transiciones especificaremos sus estados de origen y destino, la guarda y la acción (puede ser vacía).

- Una sección **strategy**, en la cual definimos regiones de inicio (obligatoria, usada para indicar el estado inicial y las precondiciones) y error (opcional). Las regiones se definen por medio de condiciones sobre las variables y el estado en el que nos encontramos (para esto último usaremos la variable *state*). En el caso de la región de inicio, indicaremos el estado inicial y los valores iniciales de las variables (en caso de que sea necesario).

A continuación mostramos un ejemplo en FAST de un autómata formado por dos estados, dos transiciones y una región inicial. Junto a él incluimos, además, su representación gráfica, en la cual las guardas y asignaciones aparecen mostradas en azul y rojo, respectivamente.

```
model ejemplo {
  var i, n;
  states s0, s1;
  transition t0 :={
   from    := s0 ;
   to      := s1;
   guard   := i < n;
   action  := ;
  };
  transition t1 :={
   from    := s1;
   to      := s0;
   guard   := true;
   action  := i' = i+1;
  };
}
strategy xx {
   Region init:={ state = s0 && i=0 && n>=0);
}
```



3.1.2. Uso de ASPIC y RANK

Antes de pasar el autómata por RANK, necesitamos incluir primero los invariantes de cada uno de sus estados. Para esto usaremos la herramienta ASPIC (generador de invariantes). Por ejemplo, si tenemos nuestro autómata en main.fst, ejecutamos los siguientes comandos:

- **aspic -ranking main.fst**: Genera los invariantes de los estados del autómata, y guarda el resultado (autómata e invariantes) en el archivo main.fstb.
- **rank main.fstb**: Calcula e imprime las funciones de rango para los estados del autómata, en caso de que los encuentre. En caso contrario, informa de que no ha sido posible. En este último caso, es posible además que haya detectado algún bucle infinito en el autómata e imprima las transiciones que lo componen.

El binario de RANK se puede obtener desde el sitio web de la *ENS de Lyon*[1], mientras que el de ASPIC se encuentra disponible en la página de *Laure Gonnord*[2].

3.2. Ejemplos

En esta sección, mostraremos cómo usamos RANK para demostrar la terminación y obtener funciones de rango de algoritmos ampliamente usados en la informática, como una inserción en un vector ordenado o la ordenación por medio del algoritmo *quicksort*, a fin de diseñar más adelante un algoritmo que realice este proceso de forma automática. Para cada uno de los ejemplos, usaremos el siguiente procedimiento general, realizando los ajustes que consideremos necesarios para probar la terminación del programa:

- Partimos del código fuente original del algoritmo a analizar.
- Creamos una versión modificada del código, en la cual las estructuras no numéricas (por ejemplo, vectores o árboles) pasan a ser resumidas mediante valores numéricos. En muchos casos, esto nos obligará a asignar valores desconocidos a algunas variables: en el caso de vectores, podemos usar una variable numérica para representar su longitud, mientras que el valor de uno de sus elementos lo sustituiremos por un valor desconocido ($a = ?$).

[1] `compsys-tools.ens-lyon.fr/rank/`
[2] `laure.gonnord.org/pro/aspic/aspic.html`

- Creamos el autómata que represente el código modificado. Trataremos de usar el menor número de estados y transiciones posibles para mantener una complejidad reducida. En caso de encontrarnos con una función que devuelva un valor, crearemos un estado final *stop* y una variable *result* en la cual almacenaremos el valor devuelto, con el fin de analizar los invariantes generados para el resultado.

- Procesamos el autómata mediante ASPIC (para obtener invariantes) y luego mediante RANK (para obtener funciones de rango). Mostraremos los resultados obtenidos a modo de tabla para aumentar su legibilidad.

3.2.1. Insert ordenado

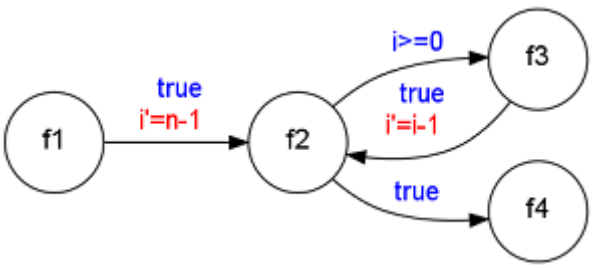
El primer ejemplo que veremos es el de una función que inserta un elemento en un vector ordenado de forma descendente. A continuación podemos ver el código, en el cual recorremos el vector de derecha a izquierda para mover una posición los elementos que sean menores que el que queremos insertar. Cuando ya no tengamos más elementos menores, insertamos x y terminamos.

```
void insertOrdenado(int A[], int n, int x) {
    int i = n-1;
    while (i >= 0 && A[i] < x) {
        A[i+1] = A[i];
        i--;
    }
    A[i+1] = x;
}
```

En el código tenemos cuatro variables: $\mathbf{A}$, n, x e i. En RANK, sin embargo, sólo podemos representar números enteros, por lo que, a la hora de crear el autómata, tenemos que descartar aquellas instrucciones que hagan uso del contenido de un vector, obteniendo así el siguiente código:

```
void insertOrdenado(int A[], int n, int x) {
    int i = n-1;
    while (i >= 0 && ?) {
        i--;
    }
}
```

En esta versión, tan sólo trabajamos con dos variables: i y n. Usaremos cuatro estados para representar el autómata: dos para los estados inicial y final, y dos para representar el bucle **while** (condición e interior del bucle):



Para representar el bucle, creamos dos transiciones desde f_2: una para cuando cumplimos la condición, y otra para representar el final del bucle. Dado que no sabemos cuándo la condición representada por **?** se deja de cumplir, supondremos que puede ocurrir en cualquier momento y, por ello, usaremos **true** como guarda desde $f2$ hasta $f4$. A simple vista, observamos que el problema ha de terminar, pues i no puede ser positivo eternamente. Tras pasar el autómata por ASPIC y RANK, obtenemos el siguiente resultado (mostrado en forma de tabla para su mejor entendimiento):

Estado	Invariantes	Función de rango
f1	• n >= 0 • n = n_0 • i = i_0	• 2 + (2 * n)
f2	• i + 1 <= n • i + 1 >= 0 • n = n_0	• 3 + (2 * i)
f3	• i + 1 <= n • i >= 0 • n = n_0	• 2 + (2 * i)
f4	• i + 1 <= n • i + 1 >= 0 • n = n_0	• 0
____f1	• n_0 >= 0	• 3 + (2 * n_0)

En la tabla podemos observar un nuevo estado $\_\_\_f1$, el cual es generado automáticamente por ASPIC como nuevo estado inicial junto a los valores iniciales de las variables (i_0 y n_0). Vemos, además, que todas las funciones de rango son unidimensionales, lo cual es de esperar en una función con un bucle con una única variable en la condición.

3.2.2. Recursión múltiple: función de Ackermann

A pesar de ser compatible con RANK, c2fsm (convertidor de lenguaje C a Integer Interpreter Automata) no soporta la traducción de funciones recursivas. Sin embargo, sí es posible generar directamente en FAST autómatas que representen estas funciones. Para conseguirlo, basta con crear múltiples transiciones desde el estado que representa una función hacia sí mismo, permitiendo que se pueda elegir más de una a la vez. Esto convertirá el autómata en no determinista, creando un camino posible por cada llamada recursiva. En el siguiente ejemplo mostraremos cómo es posible analizar la función de Ackermann, la cual presenta recursión múltiple. Comenzamos adjuntando el código de la función:

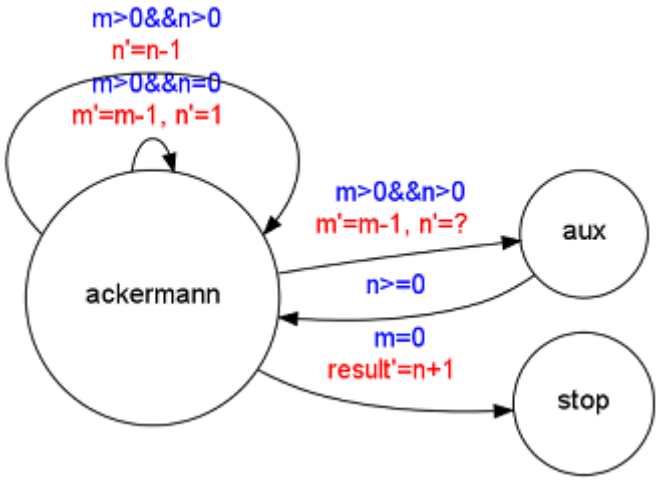
```
int ackermann(int m, int n) {
    int result;
    if (m == 0) {
        result = n+1;
    } else if (m > 0 && n == 0) {
        result = ackermann(m-1, 1);
    } else if (m > 0 && n > 0) {
        result = ackermann(m-1, ackermann(m, m-1));
    }
    return result;
}
```

En el código, podemos apreciar una condición con tres ramas. Cada una de ellas la representamos de forma distinta en el autómata:

- Una rama que devuelve directamente el resultado, cuando $m = 0$. La representamos como una transición desde el estado inicial hasta el estado *stop*, puesto que la función termina.
- Una rama con recursión final, cuando $m > 0$ y $n = 0$. Dado que conocemos el valor que toman los parámetros, la representamos como una transición del estado inicial a sí mismo.
- Una última rama con recursión múltiple (dos llamadas), en la cual uno de los parámetros de la segunda llamada es el resultado de la otra. La forma más fácil de representar esta situación

es creando dos transiciones del estado inicial a sí mismo: una de ellas con los dos parámetros asignados, y la otra con un parámetro con valor desconocido ($n' =?$, en este caso). Sin embargo, esto no es suficiente para que RANK obtenga una función de rango, ya que necesita una cota inferior del valor de n para calcularla. Dado que el parámetro n de la función ha de ser mayor o igual a 0, dividimos la transición original en dos para forzar la precondición:

- Una transición desde el estado inicial a un nuevo estado *aux*, en el cual asignaremos un valor arbitrario a n ($n' =?$). La guarda de esta transición es la misma que la original.
- Una transición desde el estado *aux* al estado inicial, la cual se tomará únicamente cuando el valor de n sea positivo. De esta forma, evitaremos volver al estado *ackermann* con valores negativos de n.



De esta forma, obtenemos el autómata de la imagen, y las funciones de rango e invariantes que mostramos en la siguiente tablam donde podemos apreciar que la función de rango que nos devuelve es una tupla lexicográfica de dimensión 2.

Estado	Invariantes	Función de rango
ackermann	• n + 1 > 0 • m + 1 > 0 • m + n + 1 > 0 • m <= m_0 • result = $result_0$	• 1 + (2 * m) • m + n
aux	• m + 1 > 0 • m + 1 <= m_0 • result = $result_0$	• 2 + (2 * m)
stop	• result > 0 • m = 0 • n + 1 = result	• 0
____ackermann	• n_0 >= 0 • m_0 >= 0	• 2 + (2 * m_0)

3.2.3. Funciones independientes: Mergesort

En este siguiente ejemplo analizaremos el algoritmo de ordenación Mergesort, el cual ordena un vector en tiempo $\mathcal{O}(n \log n)$, donde n es el número de elementos. La función principal de este algoritmo

recibe un vector y las posiciones de inicio y fin de la parte a ordenar (*start* y *end* en el código que usaremos). Tras esto, realiza las siguientes operaciones:

- En caso de que el vector tenga 1 o menos elementos, el algoritmo devuelve el vector sin modificar. De lo contrario, se sigue el resto de pasos.
- Se divide el vector en dos mitades, y se llama de forma recursiva a *mergesort* para ordenar cada una de ellas, por separado.
- La función auxiliar merge crea un nuevo vector de tamaño n ordenado con los elementos de ambas mitades. Para ello, se comienza con dos índices que señalan al primer elemento de cada una de las mitades, y se va eligiendo, en cada iteración, el menor elemento posible. Tras elegir uno de los elementos, se aumenta la posición del índice, obteniendo así un vector ordenado.

A continuación adjuntamos el código de un implementación del algoritmo, en la cual se usa una función independiente para el último paso, llamada *merge*. Dado que la terminación de esta última no depende del resto del código, podemos analizar primero su terminación y, una vez hayamos obtenido resultados satisfactorios, analizar la función principal eliminando la llamada a *merge*, pues sólo modifica los elementos del vector.

```
void mergeSort(int V[], int start, int end, int T[]) {
    if(end - start >= 1) {
        int middle = (start + end) / 2;
        mergeSort(V, start, middle-1, T);
        mergeSort(V, middle, end, T);
        merge(V, start, middle, end, T);
        for(i = start; i <= end; i++) {
            V[i] = T[i];
        }
    }
}
void merge(int V[], int start, int middle, int end, int T[]) {
    i = start, j = middle;
    for (k = start; k < end; k++) {
        if (i < middle && (j >= end || V[i] <= V[j])) {
            T[k] = V[i];
            i = i + 1;
        } else {
            T[k] = V[j];
            j = j + 1;
        }
    }
}
```

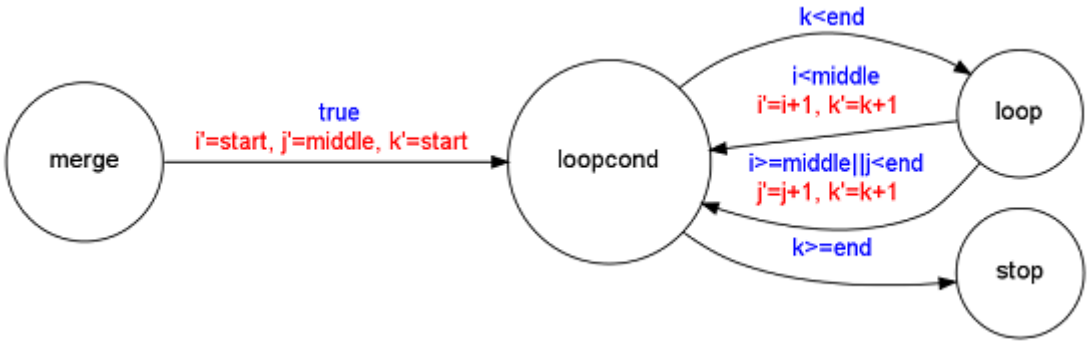
En el código C realizamos operaciones con los elementos de los vectores V y T. Dado que en los autómatas sólo trabajamos con enteros, no nos es posible representar estos valores, por lo que ignoramos por completo estas variables y sustituimos las condiciones que dependan de ellos por valores desconocidos.

Pasamos a crear el autómata de *merge*. Esta función está básicamente formada por un bucle, en el cual tenemos una estructura *if-else*. Usaremos un estado inicial, dos estados para representar el inicio y el interior del bucle, y un estado final. Dado que la condición que usamos en el *if-else* depende de un valor desconocido, perdemos precisión y habrá situaciones en las que ambas ramas serán válidas. Sin embargo, como la terminación del bucle *for* no depende del valor de i ni el de j, no corremos riesgo de crear ramas infinitas.

Tras pasar el autómata anterior por RANK, obtenemos sus funciones de rango, por lo que podemos asegurar que la parte iterativa del algoritmo termina para cualquier entrada:

Ahora nos queda analizar la función general, *mergeSort*. A simple vista, contiene una única condición, dos llamadas recursivas, una llamada a *merge* que podemos ignorar (sabemos que la llamada

```
void mergeSort(int start, int end) {
    if(end - start >= 1) {
        int middle = (start + end + 1) / 2;
        mergeSort(start, middle-1);
        mergeSort(middle, end);
        merge(start, middle, end);
        for(i = start; i <= end; i++) {}
    }
}
void merge(int start, int middle, int end) {
    int i = start, j = middle;
    for (k = start; k < end; k++) {
        if (i < middle && (j >= end || ?)) {
            i++;
        } else {
            j++;
        }
    }
}
```



termina y no afecta a ninguna de nuestras variables, por la que nos la podemos saltar) y un bucle en el que no hacemos nada. La parte complicada, sin embargo, procede de la división por 2 en la asignación a *middle*: *RANK* no soporta divisiones, por lo que tendremos que usar alguna de las siguientes alternativas:

- Ignorar la división y asignar un valor desconocido a *middle*. Sin embargo, esto haría que el autómata no terminase, pues al menos uno de los parámetros de las llamadas recursivas serían desconocidos y será posible entrar siempre en el *if*.

- Crear una transición a un nuevo estado intermedio, en la cual asignaremos un valor desconocido a *middle*. Tras esto, creamos una transición con una guarda que compruebe que el numerador de la división se encuentre en el intervalo $[2*middle, 2*middle+1]$, consiguiendo así simular la división entera por 2. Esta operación es aplicable también a otro divisor n, usando $[n*val, n*val+(n-1)]$ como intervalo, donde *val* es el numerador.

Dado que la primera opción no nos sirve, usamos la segunda: en caso de que la condición del *if* se cumpla, pasamos al estado intermedio (*s1*) y usamos una guarda para forzar a que el valor de *middle* en *s2* sea el que obtendríamos en una división entera por 2. Tras esto, continuamos por tres ramas: una rama para cada llamada recursiva y una última en la que realizamos el bucle del final y terminamos la función. Para reducir el tamaño del autómata hemos decidido obviar el estado *stop*, pero es posible añadirlo para obtener invariantes del final de la función.

Tras construir este autómata en el formato *FAST*, obtenemos las funciones de rango y concluimos que esta función termina si la llamada a *merge* lo hace. Dado que esto último lo hemos probado antes, podemos afirmar que el algoritmo termina para cualquier entrada válida.

Estado	Invariantes	Función de rango
merge	• $i = i_0$ • $end = end_0$ • $k = k_0$ • $start = start_0$ • $j = j_0$ • $middle = middle_0$	• $2 + (2 * end_0)$
loopcond	• middle <= j • start + middle + end + 1 > i + j • end >= 0 • start >= 0 • start <= i • middle + k = i + j • $start = start_0$ • $middle = middle_0$ • $end = end_0$	• ((((1 + (2 * start)) + (2 * middle)) + (2 * end)) + (- 2 * i)) + (- 2 * j)
loop	• start >= 0 • middle + end > i + j • middle <= j • start <= i • middle + k = i + j • $end = end_0$ • $middle = middle_0$ • $start = start_0$	• ((((2 * start) + (2 * middle)) + (2 * end)) + (- 2 * i)) + (- 2 * j)
stop	• middle + end <= i + j • middle <= j • start + middle + end + 1 > i + j • end >= 0 • start >= 0 • start <= i • middle + k = i + j • $start = start_0$ • $middle = middle_0$ • $end = end_0$	• 0
____merge	• $end_0 >= 0$ • $start_0 >= 0$	• $3 + (2 * end_0)$

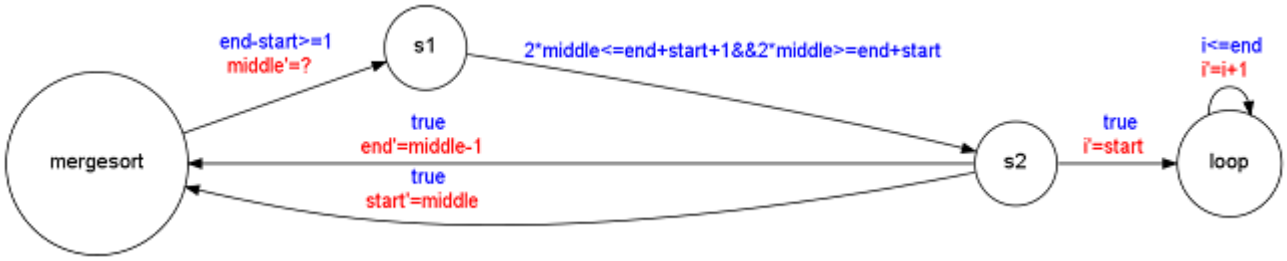
3.2.4. Ejemplo de limitaciones: Recorrido en preorden

Dado que detectar la terminación de un programa está catalogado como un *problema indecidible*, es de esperar que nos encontremos con algoritmos en los que RANK no pueda encontrar una función de rango. El recorrido de un árbol en preorden por medio de una pila es un ejemplo, el cual consiste en visitar todos los nodos del árbol nivel por nivel, de izquierda a derecha. Para conseguir esto último, obtenemos el nodo de la cima de la pila (en la cual encontraremos al principio el nodo raíz) e insertamos su hijo derecho y luego el izquierdo.

Dado que queremos transformar el código anterior en un autómata que trabaja con números enteros, tenemos que representar los tipos de datos Stack y Arbol mediante enteros. En este caso, decidimos representar Arbol como la altura del nodo raíz (donde 0 es un nodo vacío) y Stack como el número de elementos de la pila. Sin embargo, dado que no nos es posible almacenar las alturas de cada uno de los nodos que hay en la pila, no podemos saber la altura del nodo de la cima, lo cual nos lleva a pensar que RANK no tendrá suficiente información para asegurar la terminación del nuevo algoritmo.

Pasamos ahora a transformar el código a autómata. Usaremos cuatro estados: el estado inicial *start*, el estado final *stop* y dos estados para representar el bucle (hacen falta al menos dos, pues necesitamos un estado general *loop1* y un estado intermedio *loop2* al que ir tras asignar un valor arbitrario a la variable *nodo*, evitando así valores negativos de *nodo* antes de volver al primer estado).

Si observamos el autómata, vemos la posibilidad de quedarnos eternamente en el bucle (estados *loop1* y *loop2*), pues existe la posibilidad de que el valor arbitrario asignado a *nodo* sea siempre positivo. Por tanto, no es posible utilizar este autómata para demostrar la terminación del algoritmo y RANK no conseguirá encontrar ninguna función de rango. Este problema nos lo encontraremos habitualmente con



Estado	Invariantes	Función de rango
mergesort	• 3start <= 2end + 3start$_0$ + end$_0$ • end <= end$_0$ • start >= start$_0$ • i = i$_0$	• (((- 6 * start) + (4 * end)) + (6 * start$_0$)) + (2 * end$_0$)
s1	• start >= start$_0$ • end <= end$_0$ • start + 1 <= end • i = i$_0$	• (((1 + (- 4 * start)) + (2 * end)) + (6 * start$_0$)) + (2 * end$_0$)
s2	• start >= start$_0$ • end <= end$_0$ • start + 1 <= end • start + end <= 2middle • start + end + 1 >= 2middle • i = i$_0$	• (((- 4 * start) + (2 * end)) + (6 * start$_0$)) + (2 * end$_0$)
____mergesort	• end$_0$ >= 0 • start$_0$ >= 0	• 1 + (6 * end$_0$)
loop	• end + 1 >= i • start >= start$_0$ • start + end + 1 >= 2middle • end <= end$_0$ • start + end <= 2middle • start + 1 <= end • start <= i	• (1 + end) + (- 1 * i)

algoritmos cuya terminación dependa del valor de los elementos de pilas, árboles o vectores, puesto que en la transformación a enteros conservamos únicamente información relativa al número de elementos, pero no sus valores.

A pesar de lo anterior, ASPIC sigue obteniendo los invariantes de los estados, de los cuales podemos observar que el número de elementos de pila nunca será negativo (lo contrario indicaría que estamos sacando elementos de una pila vacía) y que, al terminar el programa, la pila estará vacía (ASPIC muestra que $pila \geq 0$ y $pila < 1$, pero el único valor entero que lo cumple es $pila = 0$).

3.2.5. Uso de invariantes en funciones auxiliares: quicksort y partición

Estudiaremos a continuación el ejemplo del algoritmo de ordenación quicksort, en este ejemplo introducimos una nueva idea, la necesidad de, en determinadas situaciones, conocer el invariante resultante de una función auxiliar para conocer la terminación de nuestro programa.

En primer lugar presentaremos nuestro código de quicksort y partición:

Analizando independiente partición observamos que:

- La condición de terminación se basa un bucle **while** en el que se incrementa el valor de j de manera independiente de otra condición. Es de esperar que partición de manera individual termine.
- El valor devuelto es i, que está comprendido entre $start$ y $end - 1$.

Pasamos a analizar formalmente el resultado de `partición` y a estudiar, como ya adelantamos anteriormente, su invariante en el estado *stop*.

```
void preorden(Arbol raiz) {
    Stack pila;
    pila.insert(raiz);
    while(!pila.empty()) {
        Arbol nodo = pila.pop();
        if (nodo != NULL) {
            pila.insert(nodo.der);
            pila.insert(nodo.izq);
        }
    }
}
```

```
void preoden(int raiz) {
    int pila = 0;
    pila++;
    while(pila >= 1) {
        int nodo = ?;
        pila--;
        if (nodo >= 1) {
            pila++;
            pila++;
        }
    }
}
```

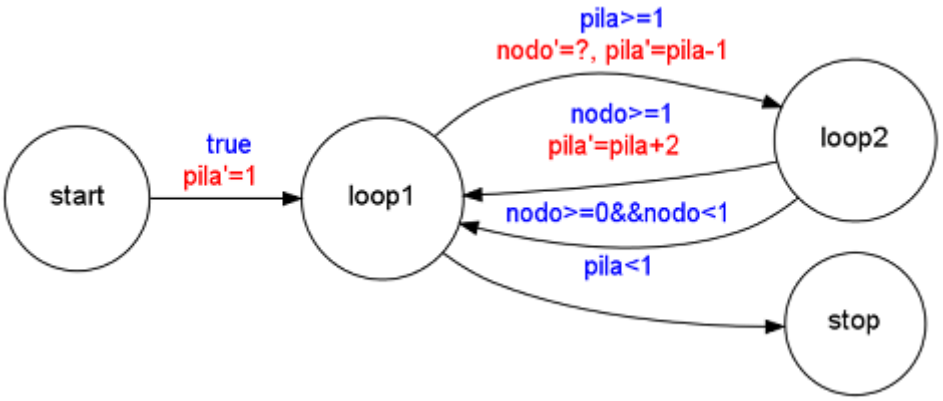
- Código:
- Autómata:
  - Tres estados: *start*,*stop* y *a* (este último intermedio, en el que se realiza el bucle).
  - En la primera transición inicializamos los valores de i y j.
  - En el bucle de *a* tenemos dos opciones ambas con guarda **true**:
    - aumentar j e i simultáneamente, opción que representa la opción en la que se toma el camino del *if*.
    - aumentar únicamente j, opción que representa el caso contrario.
- Salida RANK:

  Como podemos observar en el invariante de stop, i, que es el elemento que devuelve partición, es $< end$ e igualmente $\geq start$. Esta información será necesaria para `quicksort` como veremos a continuación. Es importante destacar que para que ASPIC obtenga estos resultados en los invariantes es necesaria la precondición $start < end$.

Una vez que `quicksort` esta estudiado, pasamos a analizar `quicksort`. En primer lugar supongamos que se desconoce el rango de valores posibles que devuelve partición, para ello asignaremos un valor cualquiera al pivote y comprobaremos como RANK no es capaz de detectar terminación.

- Código:
- Autómata: Dispondremos de tres estados, *start*, *a* y *end*. En la transición de *start* a *a* asignaremos un valor cualquiera a *pivot*, en las dos transiciones de *a* a *start* representaremos las dos llamadas recursivas a quicksort, en las que decrementaremos o aumentaremos en uno el valor de *pivot* y se lo asignaremos a *start* o *end* según corresponda.

  Como podemos observar, RANK desconoce la terminación de este automata puesto que una asignación aleatoria a *pivot* no da garantías de su valor, sin embargo, comprobemos ahora la



Estado	Invariantes	Función de rango
start	• pila = $pila_0$ • nodo = $nodo_0$	No disponible
loop1	• pila >= 0	No disponible
loop2	• pila >= 0	No disponible
stop	• pila >= 0 • pila < 1	No disponible
____start	• true	No disponible

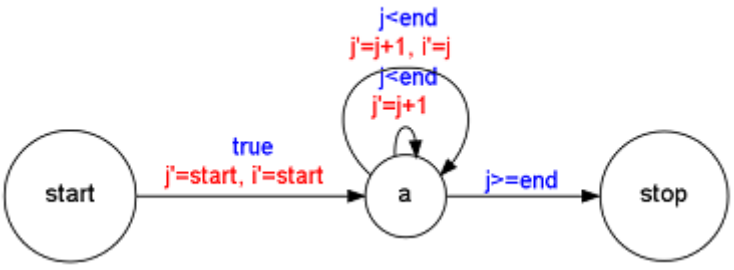
situación si le damos la información obtenida anteriormente de los invariantes de partición. El autómata entonces quedará de la siguiente manera:

No estamos seguros del valor que toma *pivot* pero sabemos que su rango se corresponde entre *start* y *stop* . Filtrando el resultado con estos valores, RANK si que es capaz de detectar terminación en este autómata obteniendo el siguiente resultado:

- Conclusión: para determinar terminación en situaciones en las que se realiza una asignación del valor de una función a una variable de la cual depende la terminación, es necesario obtener un rango de valores del invariante para poder acotar su resultado final.

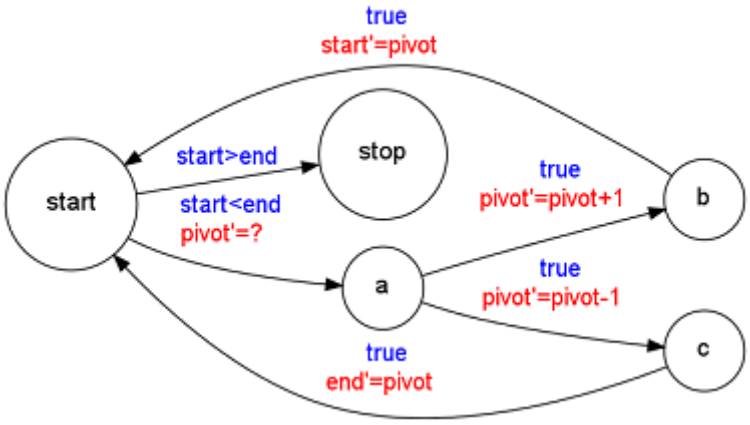
```
void quicksort(int *array, int start, int end) {
    int pivot;

    if (start < end) {
        pivot = particion(array, start, end);
        quicksort(array, start, pivot - 1);
        quicksort(array, pivot + 1, end);
    }
}


int particion(int *array, int start, int end) {
    pivot = array[i];
    i = start;
    j = start;
    while (j < end - 1) {
        if (array[j] <= i){
            swap A[i] with A[j]
            i := i + 1;
        }
        swap A[i] with A[hi]
    }
    return i;
}
```

```
int particion(int *array, int start, int end) {
    pivot = ?;
    i = start;
    j = start;
    while (j < end - 1) {
        if (? <= i){
            i := i + 1;
        }
        j++;
    }
    return i;
}
```

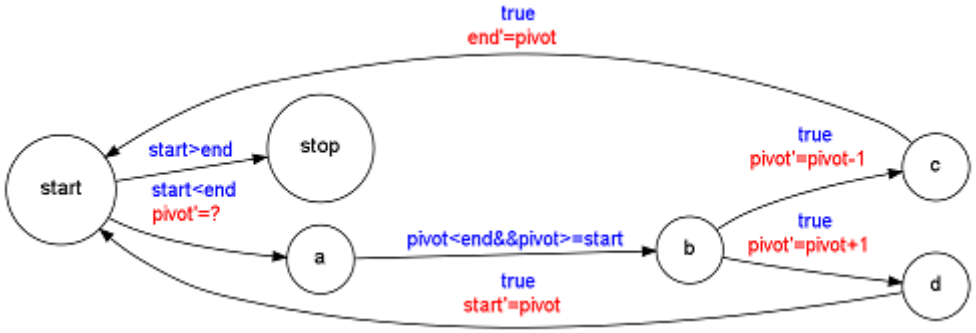


Estado	Invariantes	Función de rango
start	• start < end_0 • start >= 0 • j = j_0 • start = $start_0$ • end = end_0 • i = i_0	• (2 + (- 1 * start)) + end_0
stop	• start >= 0 • j < end + 1 • j >= end • i < end • start <= i • start = $start_0$ • end = end_0	• 0
____start	• $start_0$ < end_0 • $start_0$ >= 0	• (3 + (- 1 * $start_0$)) + end_0
a	• start >= 0 • j < end + 1 • i <= j • i < end • start <= i • end = end_0 • start = $start_0$	• (1 + (- 1 * j)) + end

```
void quicksort(int *array, int start, int end) {
    int pivot;

    if (start < end) {
        pivot = ?
        quicksort(array, start, pivot - 1);
        quicksort(array, pivot + 1, end);
    }
}
```



Estado	Invariantes	Función de rango
start		No disponible
a	• start < end	No disponible
b	• start < end	No disponible
c	• start < end	No disponible
stop	• start > end	No disponible
____start	• $start_0$ < end_0 • $start_0$ >= 0	No disponible



Estado	Invariantes	Función de rango
start	• start >= $start_0$ • end <= end_0 • start <= end + 1 • 2start < end + end_0 + 2	• ((3 + (- 4 * start)) + (2 * end)) + (2 * end_0)
a	• start >= $start_0$ • end <= end_0 • start < end	• ((2 + (- 4 * start)) + (2 * end)) + (2 * end_0)
b	• start <= pivot • start >= $start_0$ • end <= end_0 • end > pivot	• ((1 + (- 4 * start)) + (2 * end)) + (2 * end_0)
c	• end <= end_0 • end > pivot + 1 • start <= pivot + 1 • start >= $start_0$	• ((4 + (- 4 * start)) + (2 * pivot)) + (2 * end_0)
d	• end <= end_0 • end + 1 > pivot • start + 1 <= pivot • start >= $start_0$	• ((4 + (2 * end)) + (- 4 * pivot)) + (2 * end_0)
stop	• start >= $start_0$ • start > end • end <= end_0 • start <= end + 1 • 2start < end + end_0 + 2	• 0
____start	• $start_0$ < end_0 • $start_0$ >= 0	• (4 + (- 4 * $start_0$)) + (4 * end_0)

Capítulo 4

IR

4.1. IR

Nuestro programa tiene como base el lenguaje *Intermediate Representation* (IR). IR es un lenguaje funcional[5] cuya sintaxis abstracta es la siguiente:

a	$::= c$	{ constant }
	$\mid x$	{ variable }
be	$::= a$	{ atomic expression }
	$\mid f\ \overline{a_i}$	{ function/primitive operator application }
	$\mid \langle \overline{a_i} \rangle$	{ tuple construction }
	$\mid C\ \overline{a_i}$	{ constructor application }
e	$::= be$	{ binding expression }
	$\mid$ **let** $\langle \overline{x_i :: \tau_i} \rangle = be$ **in** e	{ sequential let. Left part of the binding can be a tuple }
	$\mid$ **letfun** $\overline{def_i}$ **in** e	{ recursive let for function definitions }
	$\mid$ **case** a **of** $\overline{alt_i}[;\ \_ \rightarrow e]$	{ case distinction with optional default branch }
def	$::= f\ (\overline{x_i :: \tau_i}) :: \overline{y_i :: \tau_i} = e$	{ function definition. Output results are named }
alt	$::= C\ \overline{x_i :: \tau_i} \rightarrow e$	{ case branch }
τ	$::= \alpha$	{ type variable }
	$\mid T\ \overline{\tau_i}$	{ type constructor application }

- La notación $\bar{z_i}$ es una abreviación de $z_1, ..., z_n$.
- Recursión e iteración están unificados en **let-fun**, que permite definir conjuntos de funciones mutuamente recursivas.
- Las diferentes sucesiones de asignaciones se realizan mediante el uso de expresiones **let** concatenadas.
- Ninguna variable se asigna más de una vez a un solo valor. Este dato nos resulta de especial interés, puesto que nos permitirá almacenar en una tabla el valor de cada variable sin miedo a perder información posteriormente
- Cuando una función devuelve más de un valor, devuelve una tupla de valores resultado.
- Constantes y variables se engloban en una sola clase común, atom. Esta jerarquía nos permite tratar con los literales de una función de manera conjunta y posteriormente diferenciar casos para cada tipo.

Veamos ahora el ejemplo del *quiksort*, comparado su código en IR frente a su código en C++:
Como podemos ver en el ejemplo, de dos funciones iniciales (quicksort y qsort) pasamos a tres (quicksort, qsort y f1). en las transiciones entre qsort y f1 se representa la recursión del quicksort, f1

```
void qsort(int v[],int i,int j){
  // Pre: 0 <= i <= j < length(v)
  int p;

  if (i<j){
    partition(v,i,j,p);
    qsort(v,i,p-1);
    qsort(v,p+1,j);
  }
  // Post:
  //  sorted_sub(vres,i,j+1) &&
  //  permut_sub(v,vres,i,j+1)
}

void quickSort(int v[], int n){
  // Pre: n = length(v)
  qsort(v,0,n-1);
  // Post: sorted(vres) &&
  //       permut_all(v,vres)
}
```

$quicksort(v :: array\ int, n :: int) :: (vres :: array\ int) =$
$\quad \{Q : n = length(v)\}$
$\quad \{R : sorted(vres) \wedge permut\_all(v, vres)\}$
letfun
$\quad qsort(v :: array\ int, i :: int, j :: int) :: (vres :: array\ int) =$
$\quad\quad \{Q_1 : 0 \leq i \leq j < length(v)\}$
$\quad\quad \{R_1 : sorted\_sub(vres, i, j+1) \wedge$
$\quad\quad\quad permut\_sub(v, vres, i, j+1)\}$
$\quad\quad$ **let** $(b :: bool) = <(i, j)$ **in**
$\quad\quad$ **case** b **of** $(true\ bool) \rightarrow f_1(v, i, j)$
$\quad\quad\quad\quad (false :: bool) \rightarrow v$

$\quad f_1\ (v :: array\ int,\ i :: int,\ j :: int) :: (vres :: array\ int) =$
$\quad\quad$ **let** $(v_1 :: array\ int,\ p :: int) = partition(v, i, j)$ **in**
$\quad\quad$ **let** $(p_1 :: int) = -(p, 1)$ **in**
$\quad\quad$ **let** $(v_2 :: array\ int) = qsort(v_1, i, p_1)$ **in**
$\quad\quad$ **let** $(p_2 :: int) = +(p, 1)$ **in**
$\quad\quad qsort(v_2, p_2, j)$
in
$\quad$ **let** $(n_1 :: int) = -(n, 1)$ **in**
$\quad qsort(v, 0, n_1)$

llama a partición y a cada uno de los dos casos de qsort y qsort comprueba si $i < j$ para volver a llamar a f1 o terminar la recursión.

4.1.1. CLIR

Sin embargo, nosotros no trataremos directamente con un lenguaje IR, sino con CLIR. CLIR es un lenguaje que pertenece a la familia de lenguajes de programación LISP.

La sintaxis es similar como veremos en el siguiente ejemplo, de nuevo basado en el quicksort:

```
(define quicksort ((v (array int)) (n int)) ((vres (array int)))
  (declare
   (assertion
    (precd (and (@ <= 0 (@ length v))
                (@ = n (@ length v))))
    (postcd (and (@ sorted vres)
                 (@ permut_all v vres)))))
  (letfun
      ((qsort ((v (array int)) (i int) (j int)) ((vsort (array int)))
         (declare
          (assertion
           (precd (and (@ <= 0 i j)
                       (@ < j (@ length v))))
           (postcd (and (@ sorted_sub vsort i (@ + 1 j))
                        (@ permut_sub v vsort i (@ + 1 j))))))
         (let ((b bool)) (@ < i j)
              (case b
                ((the bool true) (@ f1 v i j))
                ((the bool false) v))))
       (f1 ((v (array int)) (i int) (j int)) ((result (array int)))
         (let ((v1 (array int)) (p int)) (@ partition v i j)
              (let ((p1 int)) (@ - p (the int 1))
                   (let ((v2 (array int))) (@ qsort v1 i p1)
                        (let ((p2 int)) (@ + p (the int 1))
                             (@ qsort v2 p2 j)))))))
    (let ((n1 int)) (@ - n (the int 1))
         (@ qsort v (the int 0) n1))))
```

- Una S-expresión es un átomo o una lista de S-expresiones entre paréntesis. La transformación de S-expresiones entre IR y CLIR es la siguiente:

$$\langle sexp \rangle ::= \text{symbol} \quad | \quad \text{'('} \ \langle sexp \rangle + \ \text{')'}$$

- el símbolo *define* se utiliza para declarar una función $top - lvl$.
- pre-condiciones y post-condiciones se introducen en un *assertion* después del *define*.
- el símbolo @ se utiliza para declarar aplicación de funciones, siempre en notación prefijo.

4.2. El algoritmo IR2FSM

4.2.1. Tratamiento de tamaños y tipos de datos

El primer paso en el análisis de terminación consiste en la transformación del programa original en un programa abstracto en el cual se han producido las siguientes simplificaciones:

- Las estructuras de datos complejas como arboles, listas, pilas, etc. se sustituyen por valores enteros que representan su tamaño. Por ejemplo, una lista se sustituye por la longitud de la misma y un árbol por su profundidad.
- Cuando una estructura de datos se deconstruye mediante ajuste de patrones (pierde un elemento de la estructura original) o mediante la utilización de un deconstructor como *tail*, el componente recursivo resultante tendrá un tamaño igual al original menos uno.
- Cuando una guarda no puede ser representada por una expresión lineal con las variables del programa se abstrae a *true*.
- En una asignación $x = e$, cuando la expresión e no es lineal, se abstrae a $x =?$.

4.2.2. Tratamiento de Programas recursivos e imperativos

Imperativos

Cuando el programa fuente es puramente imperativo, la transformación a IR produce una función top-level cuyo cuerpo contiene un solo **letfun** con un conjunto de funciones mutuamente recursivas (pueden llamarse entre ellas). Cada una de estas funciones representa un estado de nuestro autómata.

En el caso de los programas fuente imperativos, todas las funciones tienen como acción final la llamada a otra de las funciones del programa, lo cual equivaldría a pasar al estado correspondiente a dicha función, o bien una tupla o atom que en nuestro autómata representará una transición a stop (puesto que la terminación esta garantizada en este caso).

La estrategia para generar el autómata es la siguiente:

- Se crearán un estado *start* (inicial) y un estado *stop* (final).
- Cada función **letfun** f dará lugar a un estado f.
- Una transición entre $f1$ y $f2$ en el autómata, corresponderá en la IR con una llamada a $f2$ en el cuerpo de $f1$
- La acción de la transición entre $f1$ y $f2$ consistirá en la transformación de los argumentos formales de $f1$ hasta que se convierten en los argumentos formales de $f2$.
- La guarda de la transición entre $f1$ y $f2$ consistirá en la unión de las diferentes condiciones que se han dado antes de realizar la llamada.

Recursivos finales y no finales

Cuando el programa fuente es recursivo, una llamada en la IR a $f1$ en el cuerpo de $f1$ se trata de la misma manera que una transición imperativa, con la salvedad de que la transición se produce de un estado a si mismo.

La principal diferencia, sin embargo, entre la representación de la IR y el autómata se produce cuando las funciones son recursivas y no finales.

Cuando se produce una llamada a una función en un **let** y a continuación la función continua, se debe tratar de manera especial dicha condición:

- Una llamada a $f2$ dentro de un **let** en el cuerpo de $f1$, corresponderá con una transición de $f1$ a $f2$ en el autómata.
- Una vez realizada esta transición, se continuará con el análisis del resto de la función, dejando anotado que el valor de la variable contenedora de la llamada a la función, por ejemplo a en el caso $leta = f2in..$ pasará a ser desconocido de ahora en adelante, $a =?$.

4.2.3. Algoritmo

El algoritmo que utilizaremos como base para nuestro programa se divide en cuatro puntos importantes:

Inicialización

- En primer lugar llamar a un procedimiento de inicialización. Este procedimiento se encarga de, partiendo de la función **top-lvl**, recorrer cada función realizando una serie de tareas:
  - Extraer el **letfun** principal.
  - Extraer resultados de la función **top-lvl**.
  - Colecionar los diferentes estados de nuestro autómata. Cada función se corresponderá con un estado además del estado inicial **start** y el estado final **stop**.
  - Extraer variables de cada una de las funciones. Solo extraeremos variables medibles como por ejemplo enteros o tamaños de un array. En este conjunto de variables no se encontrarán variables intermedias que utilizamos durante el recorrido de la función.
  - llamar a **CollectINI**, **Collect** y **crearFSM** para devolver finalmente el **fsm**.

Llamada inicial

- En segundo lugar tenemos un procedimiento que se encarga de recolectar las prototransiciones (PRTT) tanto de la función principal como del resto de funciones del programa. Una PRTT es una transición que requiere en un futuro ciertos ajustes para que sus acciones y sus guardas sean expresiones lineales. Es necesaria esta división entre la función principal y el resto de funciones, puesto que el estado del que proviene la función principal es **start** y es la única función con dicha característica.

Collect

- El metodo que se encarga de recolectar las $PRTT$ es **Collect**. Como argumentos Collect necesita:
  - Una **tabla** en la que almacenaremos los cambios que sufran las variables. Ejemplo:
    - $(let((J_1\ int))(@ + J\ (the\ int\ 1))$

    En este caso almacenaremos en la tabla que el valor de J_1 es la expresión $J + 1$
  - Un conjunto de **guardas** en el que se almacenarán las diferentes condiciones que se deben cumplir para realizar la transición de un estado a otro. Ejemplo:
    - $(case\ B_1$
      $((the\ bool\ false)\ (@f6\ VIH))...$

En este caso cuando continuemos por la rama de $(@f6\ VIH))$... almacenaremos en nuestro conjunto de guardas que se ha cumplido la condición de que B_1 es $false$

- Un estado inicial, del que proviene la llamada a esa función, y una expresión para comenzar a evaluar.

- **Collect** lanza nuevas PRTT en diferentes situaciones:

  - En el caso de encontrarse una llamada a una de las funciones del **letfun** dentro de un **let**, en cuyo caso llama a **Collect** para dicha función y continua con la siguiente expresión de la función original asignando ? a la variable contenedora de la llamada a la función (puesto que desconocemos su valor).
  - Para cada alternativa de un **case** se llama a la función **Collect** para que continue realizando el analisis. Cabe destacar en este caso, que la tabla se actualiza para dejar constancia de la condición que se ha cumplido para continuar por dicha alternativa.
  - En el caso de encontrarnos con un átomo, tupla o una función. Cualquiera de estas alternativas indica el fin de la recursión.

Cambios finales

- Por último realizamos los ajustes finales a las PRTT. Para ello:

  - Recorremos la tabla en la que hemos guardado los diferentes cambios que han sufrido las variables, empezando por las varibles finales hasta llegar a las variables iniciales.
    Este proceso produce como resultado acciones del tipo $i' = i + 1$, siendo i' una variable final y i una variable inicial.
  - Se realiza una labor similar para las guardas, estudiando que alternativas de **case** se tomarón y asignando dicho valor a la varible discriminante.

4.3. Ejemplos

Una vez diseñado y programado el algoritmo, pasamos a analizar en él sobre los ejemplos que vimos en el capítulo anterior. Para ello, partimos de códigos similares a los mostrados en los ejemplos anteriores, y los traducimos al lenguaje IR, el cual usa nuestro algoritmo como entrada para construir los autómatas correspondientes.

4.3.1. Insert ordenado

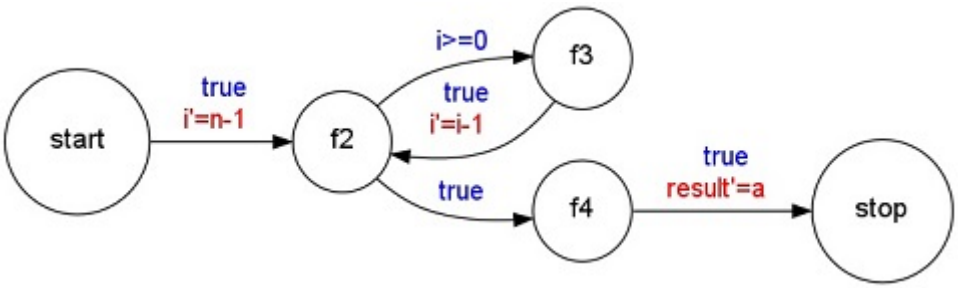
Comenzamos de nuevo por nuestro ejempolo más sencillo, el de una inserción en un vector ordenado de forma descendente. El código del que partimos es el mismo que en el capítulo 3, pero el procedimiento es bastante distinto. Procedemos primero a traducir el código al lenguaje IR, obtieniendo así cuatro funciones, correspondientes a las siguientes partes del código:

- start: inicio de la función
- f2: condición del bucle
- f3: interior del bucle
- f4: final de la función

```c
void insertOrdenado(int A[], int n, int x) {
    int i = n-1;
    while (i >= 0 && A[i] < x) {
        A[i+1] = A[i];
        i--;
    }
    A[i+1] = x;
}
```

```
start(a, n, x) =
        let i = n-1 in
        f2 (a, i, n, x)

f2 (a, i, n, x) =
        let b1 = i >= 0 in
        let e = a[i] in
        let b2 = x < e in
        let b3 = b1 ^ b2 in
        case b3 of
                true -> f3 (a, i, n, x)
                false -> f4 (a, i, n, x)

f3 (a, i, n, x) =
        let e' = a[i] in
        let i2 = i+1 in
        let a' = a[i2 <- e'] in
        let i3 = i-1 in
        f2 (a', i3, n, x)

f4 (a, i, n, x) =
        let i2 = i+1 in
        let a' = a2 [i2 <- x] in
        a'
```

Dado que el algoritmo genera al menos un estado por cada función en el código IR, es de esperar que nos encontremos al menos cinco estados: cuatro correspondientes a las funciones del código IR, y un estado stop, el cual podemos usar para extraer invariantes del resultado de la función.



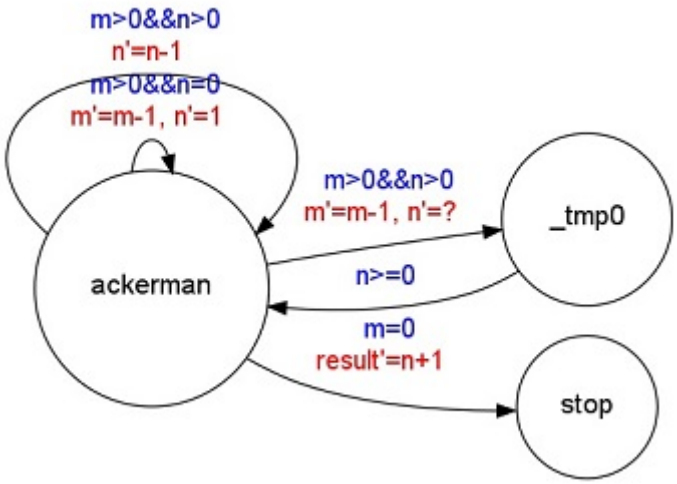
Como podemos apreciar, el autómata resultante es bastante similar al del capítulo 4.2.1, lo cual es de esperar al no haber mucha diferencia en el procedimiento usado, excepto en la traducción a IR. Debido a esto, las funciones de rango de los estados iniciales son prácticamente idénticas ($2i + 2$ en la versión manual y $2i + 3$ para la versión con IR, debido a la existencia de un estado adicional).

State	Invariants	Ranking function
start	• $n = n_0$ • $x = x_0$ • $i = i_0$ • $result = result_0$ • $a = a_0$	• $3 + (2 * n_0)$
f2	• $i + 1 >= 0$ • $i + 1 <= n$ • $a = a_0$ • $n = n_0$ • $x = x_0$ • $result = result_0$	• $4 + (2 * i)$
f3	• $i + 1 <= n$ • $i >= 0$ • $a = a_0$ • $n = n_0$ • $x = x_0$ • $result = result_0$	• $3 + (2 * i)$
f4	• $i + 1 >= 0$ • $i + 1 <= n$ • $a = a_0$ • $n = n_0$ • $x = x_0$ • $result = result_0$	• 1
stop	• $i + 1 <= n_0$ • $i + 1 >= 0$ • $n = n_0$ • $x = x_0$ • $a = result$ • $a = a_0$	• 0
____start	• $n_0 >= 0$	• $4 + (2 * n_0)$

4.3.2. Ackermann

Pasamos ahora a analizar un ejemplo con recursión múltiple: la función de Ackermann. Volvemos a partir del mismo código ya usado anteriormente, y podemos apreciar que, salvo cambios en la nomenclatura, el algoritmo es idéntico al del capítulo anterior: un estado inicial que representa la única función que tenemos, un estado auxiliar (al cual nuestra implementación le ha asignado el nombre *_tmp0*) para restringir el nuevo valor de n a los valores no negativos y un estado final *stop*, usado para obtener un invariante del resultado.

```
int ackermann(int m, int n) {
    int result;
    if (m == 0) {
        result = n+1;
    } else if (m > 0 && n == 0) {
        result = ackermann(m-1, 1);
    } else if (m > 0 && n > 0) {
        result = ackermann(m-1, ackermann(m, m-1));
    }
    return result;
}
```



Dado que los invariantes y funciones de rango son idénticos a los del anterior ejemplo de Ackermann, hemos decidido no incluirlos en este apartado. Sin embargo, se pueden consultar en el ejemplo del capítulo anterior.

4.3.3. Quicksort

En el caso del algoritmo de ordenamiento *quicksort*, separamos una vez más el código en dos partes: el código perteneciente a la función *partition* y el código principal del algoritmo. En el capítulo anterior, vimos que para detectar la terminación de este último es necesario obtener primero un invariante para el resultado de *partition* que indique que el pivote se encuentra entre el inicio y el final del subvector que analizamos. Nuestra implementación es capaz de detectar la terminación del código que mostramos a continuación, usando el siguiente procedimiento:

- Generamos el autómata correspondiente al código principal. En él, para cada llamada a una función externa (es decir, analizada por separado):
  - Teniendo en cuenta que la función recibe n parámetros y devuelve m variables como resultado, se crean $n + m$ variables temporales para guardar los valores de los parámetros y resultados de la llamada.
  - Se crea un estado intermedio (en este caso, $\_tmp0$) y una transición del estado que represente la función actual a este nuevo estado, asignando los valores correspondientes a las variables que representan los parámetros de la llamada ($\_p0$ a $\_p2$) y un valor desconocido a las que representan los resultados ($\_r0$ a $\_r1$), para más tarde restringir sus valores por medio de invariantes.
  - Las llamadas a otras funciones que se realicen a continuación saldrán desde el el nuevo estado e incluirán, como base, el invariante que hayamos obtenido para el resultado de la función, con el cual restringiremos los valores de los resultados de la llamada externa. Dado que de momento no tenemos datos sobre el resultado de *partition*, lo dejamos a *true*.

```
start (v) =
        let n = length (v) in
        let n1 = n - 1 in
        qsort (v, 0, n1)

qsort (v, i, j) =
        let b = i < j in
        case b of
        true -> f1 (v, i, j)
        false -> v

f1 (v, i, j) =
```

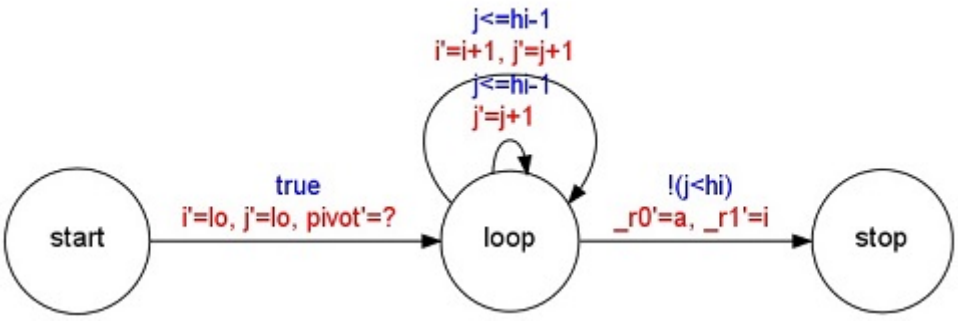
```
        let (v1, p) = partition (v, i, j) in
        let p1 = p - 1 in
        let v2 = qsort (v1, i, p1) in
        let p2 = p + 1 in
        qsort (v2, p2, j)
```

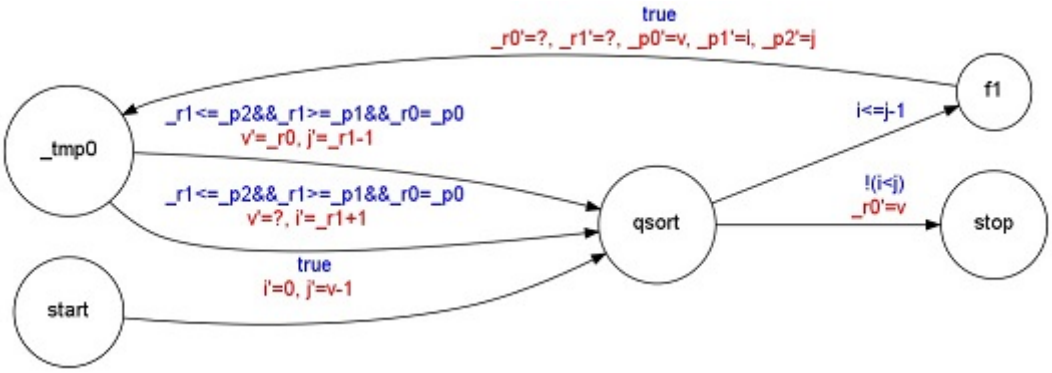
```
partition (a, lo, hi) =
        let pivot = a[hi] in
        loop_p (a lo lo hi pivot)

loop_p (a i j hi pivot) =
        let b1 = j < hi in
        case b1 of
        true -> let aj = a[j] in
                let b2 = aj <= pivot in
                let j1 = j + 1 in
                case b2 of
                false -> loop_p (a i j1 hi pivot)
                true -> let a1 = swap (a, i, j) in
                        let i1 = i + 1 in
                        loop_p (a, i1, j1, hi, pivot)
        false -> let a1 = swap (a, i, hi) in
                (a1, i)
```

- Obtenemos los invariantes de los estados que acabamos de generar, buscando propiedades entre los parámetros de la llamada externa. En este caso, obtenemos que $\_p1 <= \_p2 - 1$. Tras esto, generamos los autómatas correspondientes a cada una de las llamadas externas, usando los invariantes de los parámetros como precondiciones (esto suele requerir, además, realizar cambios en los nombres de las variables). Una vez analizados, obtenemos los invariantes de los resultados de cada autómata y los usamos como guardas de las transiciones que salgan de los estados auxiliares. Así, tras la llamada a *partition*, restringimos los valores del pivote con $\_r1 \leq \_p2$ y $\_r1 \geq \_p1$. Cabe destacar que nos interesa también probar la terminación de estas funciones externas en este paso.

- Analizamos una vez más el autómata general, para el cual RANK es ahora capaz de detectar la terminación. Por tanto, el código principal termina si la llamada externa también lo hace, y dado que al analizar *partition* hemos probado también su terminación, hemos probado entonces la terminación del programa.

Siguiendo los pasos anteriores, acabamos obteniendo los siguientes autómatas para *partition* y *quicksort* (tras incluir el invariante del anterior):





En el segundo autómata podemos apreciar el nuevo estado $\_tmp0$, el cual se crea para representar la llamada a *partition*. Las dos transiciones que salen de este estado representan las dos llamadas recursivas del algoritmo. Además, nuestra implementación es capaz de aumentar la precisión de algunas de las guardas, como en la de transición de *qsort* a *f1*), en la cual se ha sustituido la condición $i < j$ por $i \leq j - 1$ al tratarse de números enteros, lo cual ayuda a reducir el espacio de soluciones y a aumentar la precisión de los invariantes. RANK es capaz de detectar la terminación en ambos autómatas, por lo que probamos que el algoritmo termina, obteniendo además las siguientes funciones de rango para este último autómata:

Estado	Invariantes	Función de rango
_tmp0	• i >= 0 • j + 1 <= v_0 • i + 1 <= j • _p1 = i • _p2 = j • _p0 = v	• (2 + (- 3 * i)) + (3 * j)
f1	• i + 1 <= j • i >= 0 • j + 1 <= v_0	• (3 + (- 3 * i)) + (3 * j)
qsort	• i >= 0 • j + 1 <= v_0 • i <= j + 1	• (4 + (- 3 * i)) + (3 * j)
quicksort	• v >= 0 • j = j_0 • i = i_0 • _r1 = $\_r1_0$ • _r0 = $\_r0_0$ • _p2 = $\_p2_0$ • _p1 = $\_p1_0$ • v = v_0 • _p0 = $\_p0_0$	• 2 + (3 * v)
stop	• i >= j • i <= j + 1 • j + 1 <= v_0 • i >= 0 • _r0 = v	• 0
____quicksort	• v_0 >= 0	• 3 + (3 * v_0)

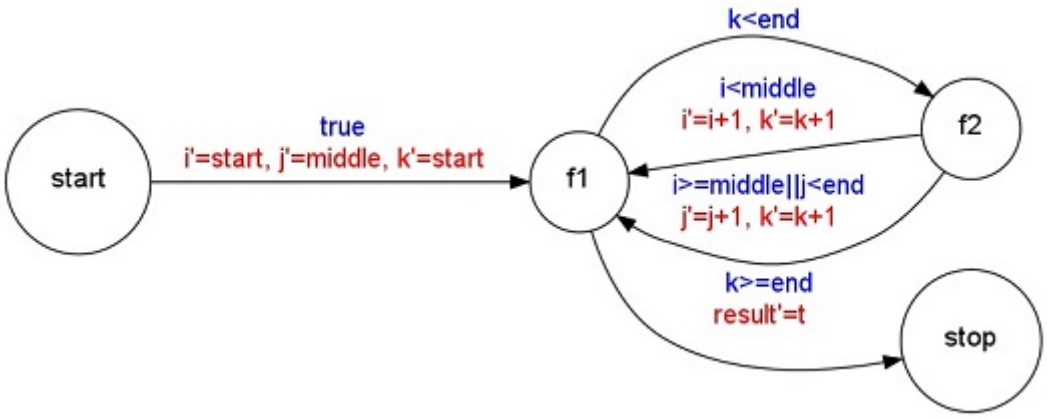
4.3.4. Mergesort

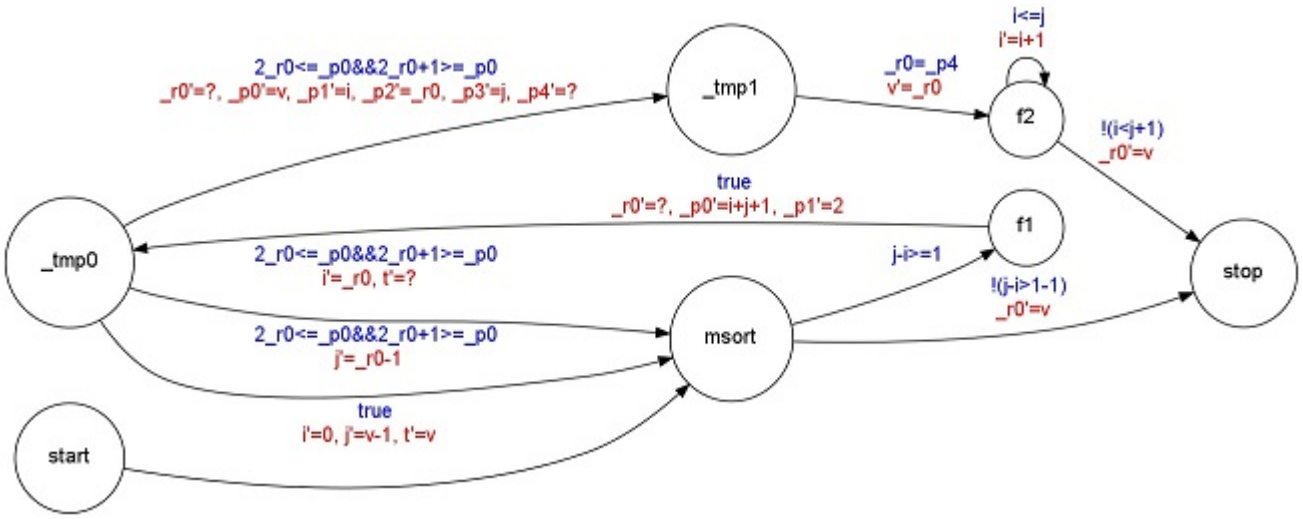
En este último ejemplo, nos volvemos a encontrar con dos funciones analizadas por separado: *merge* y *mergesort*, además del uso de una división entera, la cual, como dijimos en el capítulo anterior, no está

soportada por RANK. Para soportarla de forma fácil, tratamos a la división entera como una llamada externa a una función con dos parámetros (dividendo y divisor) y un resultado, cuyo invariante es $\_p1 * \_r0 \leq \_p0 \&\& \_p1 * \_r0 + (\_p1 - 1) \geq \_p0$. Dado que ni RANK ni ASPIC soportan operaciones no lineales (multiplicación de dos variables), sustituimos el valor del divisor, $\_p1$, a la hora de generar el autómata, obteniendo así un invariante lineal.

```
void mergeSort(int V[], int start, int end, int T[]) {
    if(end - start >= 1) {
        int middle = (start + end) / 2;
        mergeSort(V, start, middle-1, T);
        mergeSort(V, middle, end, T);
        merge(V, start, middle, end, T);
        for(i = start; i <= end; i++) {
            V[i] = T[i];
        }
    }
}
void merge(int V[], int start, int middle, int end, int T[]) {
    i = start, j = middle;
    for (k = start; k < end; k++) {
        if (i < middle && (j >= end || V[i] <= V[j])) {
            T[k] = V[i];
            i = i + 1;
        } else {
            T[k] = V[j];
            j = j + 1;
        }
    }
}
```

Tras lo anterior, el procedimiento es el mismo que en el ejemplo anterior: generamos el autómata de *mergesort* para obtener el invariante de la llamada a *merge*, luego el autómata de *merge* para obtener el invariante de su resultado (el cual resulta ser $\_r0 = \_p4$, pues modifica los valores del vector que le pasamos como parámetro) y, finalmente, volvemos a generar el autómata de *mergesort*, esta vez con el invariante anterior. Dado que RANK detecta terminación en los autómatas de ambas funciones, hemos probado la terminación de este programa.





En este último autómata, podemos apreciar los dos estados auxiliares, *_tmp0* y *_tmp1*. El primero de ellos se usa para representar la división entera y obtener el valor de *middle* (*_r0* en el autómata), e incluye tres transiciones salientes: dos transiciones a *msort*, representando las dos llamadas recursivas, y una transición a *_tmp1* para representar la llamada a *merge*.

Capítulo 5

Conclusiones

5.1. Inglés

We began this work by reviewing the state of the art of techniques that can, in certain cases, determine a program's situation. During this revision we have learnt that, while we are facing a generally undecidable problem, there are many situations in which we can automatically prove termination. It is even possible to find a complete algorithm for obtaining a certain kind of ranking functions, and this is RANK's case: when a program supports a global ranking function (i.e. a function which decreases with every transition and has the form of a lexicographic tuple of linear expressions with the program's variables), then RANK's able to find one that even has the minimum possible number of components.

The second part consisted on transitioning from a functional notation to an iterative one in the form of an automata, in order to be able to use the RANK tool on them. This way, we were able to prove termination on simple recursive functions, double recursive functions like the *quicksort* and *mergesort* algorithms and even on double recursive functions in which the result of the first recursive call is used as an argument to the second one.

The transformation algorithm is written in Java and is included as an appendix. This algorithm has now become a new component for the CAVI-ART platform.

This work has helped us on familiarizing with the CAVI-ART platform and the kind of problems that are approached on a research project, which could be useful for our professional future.

5.2. Español

El trabajo ha realizado primero una revisión del estado del arte en lo relativo a técnicas que deciden en determinados casos la terminación de los programas. En esta revisión hemos aprendido, que a pesar de tratarse de un problema indecidible en el caso general, hay muchas situaciones en las que se puede probar automáticamente la terminación, e incluso es posible encontrar un algoritmo completo para funciones de rango de un cierto tipo. Ese es el caso de RANK: si un programa admite una función de rango global, es decir una función que decrezca en cada transición del mismo, y que tenga la forma de una tupla lexicográfica de expresiones lineales en las variables del programa, entonces el algoritmo de RANK es capaz de encontrar una, que además tiene el número mínimo posible de componentes.

La segunda parte ha consistido en realizar una transformación desde una notación funcional a una notación iterativa en forma de autómata, con el fin de poder utilizar la herramienta RANK sobre dichos autómatas. Hemos conseguido demostrar de este modo la terminación de funciones simplemente recursivas, doblemente recursivas tales como los algortimos *quicksort* y *mergesort*, e incluso de funciones doblemente recursivas en las que el resultado de la primera llamada es un argumento para la segunda.

El algoritmo de transformación está escrito en Java y se adjunta como apéndice del trabajo. Dicho algoritmo ha pasado a ser un componente más de la plataforma CAVI-ART.

Esta parte del trabajo nos ha servido para familiarizarnos con la plataforma CAVI-ART y con el tipo de problemas que se abordan en un proyecto de investigación, lo cual podría sernos útil en nuestro futuro profesional.

Bibliografía

[1] Christophe Alias y col. «Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs». En: *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*. Ed. por Radhia Cousot y Matthieu Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, págs. 117-133. ISBN: 978-3-642-15768-4. DOI: 10.1007/978-3-642-15769-1_8. URL: http://dx.doi.org/10.1007/978-3-642-15769-1_8.

[2] S. Bardin y col. «Fast: Fast acceleration of symbolic transition systems.» En: Computer Aided Verification 2003, CAV03. Springer-Verlag, 2003, págs. 118-121.

[3] Paul Feautrier y Laure Gonnord. «Accelerated Invariant Generation for C Programs with Aspic and C2fsm». En: *Electr. Notes Theor. Comput. Sci.* 267.2 (2010), págs. 3-13. DOI: 10.1016/j.entcs.2010.09.014. URL: http://dx.doi.org/10.1016/j.entcs.2010.09.014.

[4] Chin Soon Lee, Neil D. Jones y Amir M. Ben-Amram. «The size-change principle for program termination». En: ed. por Chris Hankin y Dave Schmidt. ACM, 2001, págs. 81-92. ISBN: 1-58113-336-7. DOI: 10.1145/360204.360210. URL: http://doi.acm.org/10.1145/360204.360210.

[5] Manuel Montenegro, Ricardo Peña y Jaime Sánchez-Hernández. «A Generic Intermediate Representation for Verification Condition Generation». En: *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*. Ed. por Moreno Falaschi. Vol. 9527. Lecture Notes in Computer Science. Springer, 2015, págs. 227-243. ISBN: 978-3-319-27435-5. DOI: 10.1007/978-3-319-27436-2_14. URL: http://dx.doi.org/10.1007/978-3-319-27436-2_14.

[6] Andreas Podelski y Andrey Rybalchenko. «A Complete Method for the Synthesis of Linear Ranking Functions». En: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*. Ed. por Bernhard Steffen y Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, págs. 239-251. DOI: 10.1007/978-3-540-24622-0_20. URL: http://dx.doi.org/10.1007/978-3-540-24622-0_20.

[7] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999. ISBN: 978-0-471-98232-6.

Capítulo 6

Aportaciones

6.1. Álvaro Mínguez

Durante los primeros meses de curso, tanto Jakub como yo hemos realizado una tarea de investigación para poder familiarizarnos con los conocimientos necesarios para abordar este trabajo.

Desde Septiembre hasta Marzo, aproximadamente, hemos tenido una reunión con Ricardo una vez por semana de una duración aproximada de una hora y media en la cual debatíamos problemas que surgían en reuniones previas y avanzábamos simultáneamente en nuestro estudio.

Paralelamente a estas reuniones tanto Jakub como yo, dividiéndonos la carga de trabajo al 50

Tras finalizar este primer estudio y una vez que disponíamos de los conocimientos necesarios para abordar temas mas complicados, nos centramos en el estudio de RANK. Nuestro objetivo era formular una serie de ejemplos que, sin dejar de ser comprensibles para el lector, abarcaran el amplio espectro de la problemática a la que nos enfrentamos. Durante esta fase continuamos con nuestras reuniones semanales y de nuevo la carga de trabajo, en mi opinión, se dividió al 50

Una vez finalizado este trabajo, que paso a formar parte de nuestro trabajo como el capitulo 3, pasamos a abordar lo que es en esencia nuestra principal contribución a la IR, el programa IRtoFSM. En la realización de este programa, mi contribución se centro en la transformación de prototransiciones a transiciones (Resuelve y ResuelveAcción). Terminamos este programa con no poco esfuerzo a mediados de Junio y puesto que la memoria no estaba en el estado que buscábamos para su presentación decidimos posponer nuestra entrega hasta Septiembre.

La memoria y las diferentes revisiones que hemos realizado sobre la misma, las hemos realizado dividiéndonos la carga de trabajo de manera que fuera lo más equitativo posible y puesto que hemos realizado diferentes revisiones sobre cada uno de las secciones y sub-secciones me sería imposible diferenciar que capítulos corresponden a cada uno.

6.2. Jakub Holubansky

Casi todas las semanas (aunque con un ligero parón debido a la época de exámenes), he tenido reuniones junto a Álvaro y Ricardo en las que hablábamos de nuestros avances desde la última reunión, avanzábamos entre los tres en el trabajo y nos marcábamos las tareas a hacer para la siguiente reunión.

En los primeros meses del curso (hasta marzo, aproximadamente), nuestros avances en el trabajo consistieron en investigación del estado del arte de los algoritmos de detección de terminación. Las tareas consistieron en leer y analizar las publicaciones sobre los métodos más potentes que encontrábamos. Álvaro y yo incluimos nuestros apuntes en pequeñas *fichas* que acabaron sirviendo de base para el capítulo 2 de la memoria. La carga de trabajo nos la dividimos de forma equitativa, pues o nos centrábamos en hacer cada uno la mitad de una ficha, o hacía uno una ficha mientras el otro redactaba la siguiente.

Una vez habíamos acabado con los artículos, coincidimos todos en que RANK sería la herramienta que usaríamos en las siguientes partes del trabajo, y pasamos a realizar ejemplos manuales en los que creábamos autómatas a partir de algoritmos conocidos. Mientras yo hacía ejemplos para algoritmos como *insertsort* o *mergesort*, Álvaro construyó ejemplos para *Ackermann* y *quicksort*, pues nos

repartimos otra vez el trabajo.

Tras estos ejemplos, de los cuales los mencionados pasaron a formar parte del capítulo 3 de la memoria, comenzamos a plantear un algoritmo general que nos permitiese convertir programas de la RI de la plataforma CAVI-ART al formato de autómatas de RANK. El algoritmo fue diseñado entre los tres en las reuniones y lo programamos diviéndonos las clases Java que íbamos a programar, eligiendo yo la parte más general del algoritmo y Álvaro la dedicada al procesado de variables, guardas y asignaciones.

Finalmente, tras haber probado el funcionamiento del programa, realizamos el capítulo 4 de la memoria. La primera mitad del capítulo está hecha en su mayoría por Álvaro, mientras que yo me centré más en la segunda parte, principalmente la parte de los ejemplos.